University of Massachusetts Amherst
ScholarWorks@UMass Amherst

Doctoral Dissertations | Dissertations and Theses

October 2021

COST-EFFICIENT RESOURCE PROVISIONING FOR CLOUD-ENABLED SCHEDULERS

Lurdh Pradeep Reddy Ambati
University of Massachusetts Amherst

Follow this and additional works at: https://scholarworks.umass.edu/dissertations_2

 Part of the Computational Engineering Commons, and the Computer Engineering Commons

Recommended Citation

Ambati, Lurdh Pradeep Reddy, "COST-EFFICIENT RESOURCE PROVISIONING FOR CLOUD-ENABLED SCHEDULERS" (2021). *Doctoral Dissertations*. 2258.
https://doi.org/10.7275/24161492 https://scholarworks.umass.edu/dissertations_2/2258

This Open Access Dissertation is brought to you for free and open access by the Dissertations and Theses at ScholarWorks@UMass Amherst. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

COST-EFFICIENT RESOURCE PROVISIONING FOR CLOUD-ENABLED SCHEDULERS

A Dissertation Presented

by

LURDH PRADEEP REDDY AMBATI

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2021

Electrical and Computer Enginering

© Copyright by Lurdh Pradeep Reddy Ambati 2021

All Rights Reserved

COST-EFFICIENT RESOURCE PROVISIONING FOR CLOUD-ENABLED SCHEDULERS

A Dissertation Presented

by

LURDH PRADEEP REDDY AMBATI

Approved as to style and content by:

David E Irwin, Chair

Prashant Shenoy, Member

Lixin Gao, Member

Tongping Liu, Member

Christopher V. Hollot, Department Head
Electrical and Computer Enginering

DEDICATION

To my parents and my friends.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Prof. David Irwin, for his excellent advice and constant support, and for being an endless source of motivation throughout my doctorate studies. His patience, invaluable feedback, and perspectives on research forged every element of this dissertation. David taught me so much about the various aspects of research: finding interesting problems, keeping the bigger picture in mind, avoiding premature optimizations, and the importance of documentation. None of this would have been possible without you.

Next, I would like to thank my research mentors and colleagues. Prashant Shenoy has provided continuous guidance and close mentorship over the years. I am immensely grateful for his support. My colleagues Noman Bashir, Srini Iyengar, Akansa Singh, Supreeth Shastri, Dong Chen, and Zeal Shah took the challenging process of spending over four years working long-winded hours and made it truly fun. I'd also like to thank my dissertation committee members, Prof. Lixin Gao, and Prof. Tong Ping, for their interest in my work and their comments and feedback.

My time in Amherst has been made better by my close friends. My sincere thanks to all my friends: Sourabh Kulkarni, Subrahmanyam Nukkala, Sachin Bhat, Parth Gandhi, and Sukhneet Kaur, for giving me company through peace, joy, and panic. Special thanks to my gaming friends: Faux, Auto, Binky, Crelder, and Ant for the laughs and memorable moments.

Finally, I would like to thank my parents, my sister, and my brother-in-law, for their unwavering support and understanding. I could never have finished my Ph.D. without your support and encouragement.

ABSTRACT

COST-EFFICIENT RESOURCE PROVISIONING FOR CLOUD-ENABLED SCHEDULERS

SEPTEMBER 2021

LURDH PRADEEP REDDY AMBATI

B.E., CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor David E Irwin

Since the last decade, public cloud platforms are rapidly becoming de-facto computing platform for our society. To support the wide range of users and their diverse applications, public cloud platforms started to offer the same VMs under many purchasing options that differ across their cost, performance, availability, and time commitments. Popular purchasing options include on-demand, reserved, and transient VM types. Reserved VMs require long time commitments, whereas users can acquire and release the on-demand (and transient) VMs at any time. While transient VMs cost significantly less than on-demand VMs, platforms may revoke them at any time. In general, the stronger the commitment, i.e., longer and less flexible, the lower the price. However, longer and less flexible time commitments can increase cloud costs for users if future workloads cannot utilize the VMs they committed to buying.

Interestingly, this wide range of purchasing options provide opportunities for cost savings. However, large cloud customers often find it challenging to choose the right

mix of purchasing options to minimize their long-term costs while retaining the ability to adjust their capacity up and down in response to workload variations. Thus, optimizing the cloud costs requires users to select a mix of VM purchasing options based on their short- and long-term expectation of workload utilization. Notably, hybrid clouds combine multiple VM purchasing options or private clusters with public cloud VMs to optimize the cloud costs based on their workload expectations.

In this thesis, we address the challenge of choosing a mix of different VM purchasing options in the context of large cloud customers and thereby optimizing their cloud costs. To this end, we make the following contributions: (i) design and implement a container orchestration platform (using Kubernetes) to optimize the cost of executing mixed interactive and batch workloads on cloud platforms using on-demand and transient VMs, (ii) develop simple analytical models for different straggler mitigation techniques to better understand the cost of synchronization in distributed machine learning workloads and compare their cost and performance on on-demand and transient VMs, (iii) design multiple policies to optimize long-term cloud costs by selecting a mix of VM purchasing options based on short- and long-term expectations of workload utilization (with no job waiting), (iv) introduce the concept of waiting policy for cloud-enabled schedulers, and show that provisioning long-term resources (e.g., reserved VMs) to optimize the cloud costs is dependent on it, and (v) design and implement speculative execution and ML-based waiting time predictions (for waiting policies) to show that optimizing job waiting in the cloud is possible without accurate job runtime predictions.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Cloud computing platforms are rapidly becoming de-facto computing platforms for our society. Early providers include Amazon Elastic Compute (EC2) which offered Infrastructure-as-a-Service (IaaS) virtual machines starting in August 2006 [25]. These IaaS cloud platforms provide numerous benefits like access to compute resources on-demand, pay as you use model, an illusion of infinite scalability, and all of these without a significant upfront cost. As a result, cloud computing platforms have become ubiquitous for providing the access to large-scale computing power to nearly every field like scientific, entertainment, finance, etc. Given this widespread adoption, the global cloud computing market size is expected to grow from $371.4 billion in 2020 to $760.98 billion by 2025 [10].

With the advent of cloud computing, large institutions that have traditionally operated large private compute clusters for their general computing needs have begun to migrate to public Infrastructure-as-a-Service (IaaS) cloud platforms like Amazon AWS, Google GCP, and Microsoft Azure. These private clusters typically manage a fixed number of computing resources typically sized for peak demands, and they often have low average utilization (<30%), but may periodically experience large bursts in job arrivals, e.g., due to deadlines, product releases, or seasonal variations, that result in long job waiting times. As with large institutions, software systems like container orchestration platforms and job schedulers that manage these private computing clusters are now cloud-enabled and support dynamic cloud resources.

As large institutions migrate their workloads to the cloud platforms, they have numerous options for optimizing their cost and reduce the job waiting times. For example, schedulers may provision cloud VMs on demand to service each job, requiring them to only pay for resources when jobs need them. In this case, the cloud's operating costs are often much lower than the capital cost of an under-utilized fixed-size cluster, since the latter must effectively "pay" even when resources are idle.

1.1 Motivation

Migrating a largescale private computing infrastructure to a public cloud is a complex task. For example, the major hyper-scale public cloud providers — Amazon, Google, and Microsoft — now offer dozens of VM types with different CPU, memory, I/O, and network characteristics at different prices [9]. Thus, selecting the "right" type of VM that yields the desired performance at the lowest cost — for a particular workload can be challenging. In addition, a significant fraction of a private cluster's cost is due to upfront capital expenses, e.g., server hardware, building space, supporting IT equipment, etc., that are fixed, while a cloud-based cluster's cost is largely operational expenses that are dependent on expectations of the future workload. If cloud resources are provisioned judiciously by choosing the optimal mix of VM options when migrating large workloads to the cloud, cloud costs can be often lower than operating a private cluster. However, the large number of cloud VM configurations, the uncertainty in future workload characteristics, and the complexity of decision making when operating a large cloud cluster imply there is no guarantee that the potential cost savings from migrating to the cloud will actually be realized.

In addition, cloud platforms offer many types of VMs under a variety of different purchasing options, which differ in their cost, performance, and availability. The specific set of purchasing options, as well as their names, prices, and some details, differ across cloud providers, which we discuss in §2.2. In particular, on-demand,

reserved, and transient VMs are popular purchasing options. On-demand VMs are the most popular, enabling users to request and release VMs at any time and only be charged for the time they hold them. In contrast, the reserved option requires users to commit to buying 1 or 3 years of VM time in advance, but at a significant discount compared to holding an on-demand VM for 1 or 3 years. Of course, if reserved VMs are utilized less than their discount factor, then they will incur a higher cost than using the equivalent on-demand VMs, since users may release on-demand VMs when not in use.

Given this wide variety of VMs with different performance profiles and different purchasing options, optimizing for long-term cloud costs not only requires selecting the right VM types based on a workload's requirements, but also selecting the right mix of purchasing options based on future workload expectations. While the former problem of choosing cloud VM types has been the subject of much research [16,47,80, 88], but optimizing the purchasing options for the chosen VMs has not seen as much attention. This problem requires cloud users to balance tradeoffs — (i) making the longest possible commitments for the provisioned VMs to extract the greatest savings, while retaining some ability to make short-term increases or decreases to provisioned VMs to respond to changing workloads and (ii) using the unreliable transient VMs to significantly reduce the cost, while providing the same performance and availability guarantees as on-demand or reserved VMs.

1.2 Summary of Contributions

In this thesis, we hypothesize that, by judiciously selecting VM purchasing options to match the workload requirements, we can significantly optimize the cloud costs while mitigating the impact on performance, or vice versa. As part of our thesis, we identify the following significant problems and model the solution using classic

statistics and implement these solutions in real-world job schedulers (when applicable) to illustrate the benefits of the proposed solutions.

1.2.1 Mixed Interactive and Batch Workloads on Transient VMs

Container Orchestration Platforms (COPs), such as Kubernetes, Mesos, and others, have evolved into de facto cluster "operating systems" by automating the deployment of distributed applications encapsulated in containers, and managing the allocation of resources between them. Thus, COPs must support the availability and performance requirements of a wide range of applications, including long-lived interactive services and non-interactive batch jobs, while also maintaining high cluster utilization.

COPs were originally developed for managing a mostly static set of dedicated physical machines in data centers. However, increasingly, the resources that underly COPs are virtual machines (VMs) dynamically acquired from cloud platforms. These platforms offer many types of VMs under a variety of different contracts, which differ in their cost, performance, and availability. In particular, transient VMs are an increasingly popular VM type, since they typically cost 50-90% less than on-demand VMs. However, cloud platforms reserve the right to reclaim transient VMs at any time to satisfy higher priority tasks. Thus, while transient VMs' low price is attractive, their unreliability makes them unsuitable for COPs that must support long-lived interactive services with high availability requirements. As a result, prior work focuses primarily on optimizing only batch workloads for transient VMs.

To address the problem, we design TR-Kubernetes, a transient-aware COP that supports both batch jobs and interactive services with high availability requirements at a low cost using transient VMs. We show that TR-Kubernetes requires minimal extensions to Kubernetes, and is capable of lowering the cost (by 53%) and improving

the availability (99.999%) of a representative interactive/batch workload on Amazon EC2 when using transient compared to on-demand VMs.

1.2.2 Distributed ML Workloads on Transient VMs

Cloud platforms often execute parallel batch applications, such as distributed machine learning (ML), that include numerous synchronization barriers. These barriers, which prevent any task from advancing beyond a specified point until all tasks have reached that point, significantly degrade application performance by reducing it to that of the slowest "straggler" task. To address the problem, researchers have proposed numerous straggler mitigation techniques, including speculatively re-executing straggler tasks and various relaxations of strict barrier semantics. While these techniques improve parallel application performance, they incur a cost in terms of the resources wasted re-executing tasks or waiting. Importantly, these costs, which are often implicit in prior work that targets dedicated resources, become *explicit* in the cloud, which charges for resources at fine-grained intervals. In addition, the cost difference between techniques is exacerbated in cloud platforms, since they charge *substantially less* for transient resources that effectively yield a probabilistic performance across a wide range.

While transient resources' low list price is attractive, revocations increase the frequency and severity of stragglers, which decreases parallel job performance and increases overall execution cost. *To better understand the cost of synchronization, we develop simple analytical models of different straggler mitigation techniques and compare their cost and performance on on-demand and transient resources.* Our analysis shows that i) transient servers offer complex tradeoffs compared to on-demand servers, and can result in higher overall costs despite their highly discounted price due to their probabilistic performance; ii) common approaches to straggler mitigation, which is a well-studied problem, are less effective using transient servers that cause

frequent and severe stragglers; and iii) a recent approach to flexible synchronization offers the best cost and performance.

1.2.3 Optimizing Long-term Cloud Costs by Mixing VM Purchasing Options

Cloud platforms offer the same VMs under many purchasing options that specify different costs and time commitments, such as on-demand, reserved, sustained-use, scheduled reserve, transient, and spot block. In general, the stronger the commitment, i.e., longer and less flexible, the lower the price. However, longer and less flexible time commitments can increase cloud costs for users if future workloads cannot utilize the VMs they committed to buying. Large cloud customers often find it challenging to choose the right mix of purchasing options to reduce their long-term costs while retaining the ability to adjust capacity up and down in response to workload variations.

To address the problem, we design policies to optimize long-term cloud costs by selecting a mix of VM purchasing options based on short- and long-term expectations of workload utilization. We consider a batch trace spanning 4 years from a large shared cluster for a major state university system that includes 14k cores and 60 million job submissions, and evaluate how these jobs could be judiciously executed using cloud servers using our approach. Our results show that our policies incur a cost within 41% of an optimistic optimal offline approach, and 50% less than solely using on-demand VMs.

1.2.4 Optimal Fixed Resource Provisioning for Cloud-Enabled Schedulers

As job schedulers migrate to the cloud, they have many options for optimizing cost and reducing job waiting times. For example, schedulers may provision cloud VMs on demand to service each job, requiring them to only pay for resources when jobs need

them. Importantly, however, buying fixed resources (or reserving them for long periods) is significantly cheaper than renting resources on demand if the fixed resources are highly utilized. Thus, a mixed infrastructure that satisfies some baseload with highly-utilized fixed resources, and satisfies load bursts using on-demand resources can decrease cost. Notably, hybrid clouds, which combine fixed private resources with cloud bursting, use this approach, as do many companies, which both buy reserved VMs and dynamically rent on-demand VMs.

In this work, we introduce the concept of a waiting policy for cloud-enabled schedulers and show that provisioning fixed resources to optimize cost is dependent on it. A waiting policy is the dual of a scheduling policy: while a scheduling policy determines which jobs run when fixed resources are available, a waiting policy determines which jobs wait for fixed resources when they are not available (rather than run immediately on on-demand resources). For cloud-enabled schedulers, the waiting policy is important, since it dictates the tradeoff between job performance and cost. Our evaluation on a year-long production batch workload consisting of 14M jobs run on a 14.3k-core cluster and show that a compound waiting policy decreases the cost (by 5%) and mean job waiting time (by 7×) compared to a fixed cluster of the current size.

1.2.5 Data-driven Job Scheduling for Cloud-Enabled Schedulers

Cloud-enabled scheduling differs from conventional scheduling on fixed resources in that cost, in addition to job waiting time, is a critical metric. As a result, cloud-enabled schedulers must not only define a scheduling policy, which selects which jobs run when fixed resources become available, but also a *waiting policy*, which selects which jobs wait for fixed resources, and for how long when they are not available before running on on-demand resources. Importantly, as with many scheduling policies, optimizing the waiting policies above requires *a priori* knowledge of job runtimes.

Unfortunately, scheduling policies that require knowing job runtimes, such as shortest job first (SJF), are often not widely used because accurately predicting job runtimes remains challenging. Recent work highlights many reasons for the low prediction accuracy, including a lack of sufficient features for training machine learning (ML) models and non-stationarity in workloads that leads to inconsistent performance [53]. Directly implementing the waiting policies above suffers from the same challenges.

The main contribution of this work is showing that optimizing waiting policies for cloud-enabled schedulers is possible without accurate job runtime predictions, and can come close to the cost and waiting time achievable given perfect knowledge of job runtimes. To do so, we develop two techniques (speculative execution and ML-based waiting time predictions) to optimize job waiting under policies, respectively. Intuitively, optimizing these waiting policies in the cloud is simpler than optimizing scheduling policies for fixed resources because i) there is no hard resource constraint, and ii) our waiting policy predictions require only binary classification, i.e., where a job's running or waiting time crosses a threshold, which does not require absolute model accuracy. Our evaluation on a year-long production batch workload shows that using our techniques yields a cost and waiting time within 4% and 13%, respectively, of using waiting policies with perfect knowledge of job runtime.

1.3 Dissertation Outline

We organize the rest of the proposal as follows. Chapter 2 provides the necessary background on cloud platforms, service contracts, and container orchestration platforms (COP). Chapter 3 presents TR-Kubernetes, a COP that optimizes the cost of executing mixed interactive and batch workloads on cloud platforms using transient VMs. Chapter 4 deals with understanding synchronization costs for Distributed ML workload on transient and on-demand cloud resources. Chapter 5 addresses the challenge to choose the right mix of purchasing options to reduce the long-term cloud

costs with no job waiting. Chapter 6 describes the concept of waiting policy to optimally provision the fixed (or reserved) resources to optimize cloud costs and reduce job waiting. Chapter 7 describes how to implement waiting policies in a real-world scenario using ML-based wait time predictions and speculative execution. Finally, chapter 8 concludes the work.

CHAPTER 2

BACKGROUND

In this chapter, we will discuss the background required to understand various aspects of this dissertation. First, we discuss job schedulers in the context of this thesis. Second, we provide details on the cloud VM purchasing options and compare them to highlight the differences in their cost, required time commitment, and resource guarantees. Third, we briefly discuss different workload characteristics considered in this dissertation. Finally, we provide an overview of container orchestration platforms (COPs) like Kubernetes, Mesos, etc which are used for implementing our proposed solutions.

2.1 Job Schedulers

Job schedulers have been one of the central components of the IT infrastructure since the early computing systems. Job scheduling can be defined most simply as the orderly, reliable, sequencing of batch program execution. Examples of early job schedulers include IBM's OS/360, which had primitive capabilities for transitioning between jobs. Traditionally, these job schedulers manage a fixed number of computing resources in a private cluster at large scales. To submit jobs, users specify job resource requirements to these schedulers, which either allocate idle resources to execute them or force them to wait for idle resources to become available. Since the private clusters manage a fixed number of computing resources typically provisioned for peak demands, they often have low average utilization (<30%), but may period-

ically experience large bursts in job arrivals e.g., due to deadlines, product releases, or seasonal variations, that result in long job waiting times.

Cloud-Enabled Schedulers. As cloud platforms started providing access to large-scale computing power for nearly every enterprise sector, job schedulers began to support the cloud resources. Using these cloud resources, job schedulers have many options for optimizing cost and reducing job waiting times. For example, schedulers may provision cloud VMs on demand to service each job, requiring them to only pay for resources when their jobs need them. In this case, the cloud's operating costs are often much lower than the capital cost of an under-utilized fixed-size cluster. In addition, since the cloud provides the illusion of infinite scalability, jobs never need to wait for resources, as schedulers can always acquire cloud resources to service them immediately. Most modern job schedulers (and workload management platforms) like IBM's LSF, Kubernetes, Mesos, etc., are now cloud-enabled and support such "auto-scaling", which acquires cloud VMs to service jobs, and release them when done [8,30]. Notably, the cloud-enabled schedulers can manage both the fixed resources in a private cluster and the cloud resources at the same time. Such mixed infrastructures are referred to as "hybrid clouds", where they execute some baseload with the fixed resources and satisfy the additional load bursts using cloud resources.

2.2 Cloud Purchasing Options

Our work focuses specifically on VM purchasing options that relate to time commitments and flexibility, and not *VM types* or *capacity reservations*. Since these options only differ in the resources they offer, users can treat them as a different resource type. We also do not consider capacity reservations, which enable users to pay to ensure that their future requests for on-demand VMs are not rejected due to a lack of capacity. Currently, these capacity reservations incur the same cost as the on-demand option, so users may just as well purchase and hold on-demand VMs.

Purchasing Option	Relative Cost (%)	Time Commitment	Revocable	Cloud Providers
On-demand	100%	None	No	All
Reserved	60%	1yr	No	All
Reserved	40%	3yrs	No	All
Transient	20-40%	None	Yes	All
Sustained-Use	70-100%	None	No	Google
Customized	105%	-	-	Google
Spot Block	55-70%	None	After 1-6hrs	Amazon
Scheduled Reserved	90-95%	1200hrs-8760hrs (1yr)	No	Amazon

Table 2.1: Overview of the primary VM purchasing options across the major cloud providers

Table 2.1 lists the different VM purchasing options that we consider and their primary attributes. As can be seen, the *same* cloud VMs can be procured under a number of different purchasing options. The relative cost represents the percentage cost relative to the on-demand cost per unit time for the equivalent VM type and is *not* the percentage discount. The time commitment is the amount of time the user must commit to buying. We discuss the other attributes below.

On-demand. The on-demand option is the most common one offered by all cloud providers and typically the default option for users. As a result, we represent the cost of the other options relative to the on-demand option. An on-demand VM incurs a cost per unit time from the time the cloud platform allocates it to the user until the time the user terminates it. The per unit time cost is now billed at fine-grained resolutions, e.g., either per-second or per-minute, rather than hourly. Cloud platforms do not generally revoke on-demand VMs, but they are not guaranteed to be available when requested. That is, cloud platforms may reject users' request for on-demand VMs if they run out of data center capacity. However, the frequency of out-of-capacity rejections is not publicly known.

Reserved. The reserved option enables users to commit to buying a VM for 1- or 3-year period in return for a discount compared to procuring an on-demand VM over the same period. All cloud providers offer 1- and 3-year reserved options, which

are designed for cheaply satisfying a user's expected base load—the minimum level of demand—over the reservation's term. While reserved VMs are not revocable, they do generally guarantee the user capacity on request. That is, if a user ever terminates a reserved VM, when they request the VM later (within the reservation's term), unlike with on-demand VMs, the cloud platform guarantees to have the capacity to satisfy that request.

The reserved option is essentially a volume discount, where the actual discount is based on the time commitment as well as other options, such as the amount of upfront payment, whether reserved VMs can be "converted" to other VMs of a different type (but the same resources), and whether the reserved VMs can switched between different data centers within the same geographical region. The costs in Table 2.1—60% and 40% of the on-demand price for 1- and 3-year terms—are typical discounts for standard options, e.g., payment in full for non-convertible VMs tied to one availability zone. In this case, if the reserved VM (3yr) is utilized >40% of the time, its effective price and 3-year cost are less than the on-demand VM, thereby making it the cheaper option. We call this the break-even point.

Transient. All cloud providers offer their surplus capacity in the form of transient VMs [70] but under different names and slightly different terms. Transient VMs are the cheapest purchasing option, costing 20-40% of the on-demand cost, and come with no time commitment. However, since transient VM resources represent spare capacity, cloud platform's may revoke them at any time to satisfy higher-priority requests for on-demand and reserved VMs. Given their low cost and priority, transient VMs are not guaranteed, and requests for such VMs are likely rejected due to fluctuating surplus capacity more frequently than on-demand VMs (although the rejection rates are not publicly known). Transient VMs are generally designed for cheaply satisfying batch jobs that run in the background and can tolerate delays due to unexpected revocations.

Sustained-Use. Google offers a sustained-use discount that automatically applies to on-demand VMs of any type that are run for some fraction of a month-long billing period. The discount applies separately to each core, i.e., vCPU, and gigabyte (GB) of memory regardless of type, since Google separately charges for each core and GB of memory. Thus, VM types simply incur a cost based on their pre-defined number of cores and memory allotment. The discount starts once each core or GB of memory is used for 25% of the month and increases the longer they are used with the maximum discount being 30% off the on-demand cost for the entire month. Specifically, for the first 25% of the month users pay 100% of the on-demand cost, next for 25-50% they pay 80%, then for 50-75% they pay 60%, and finally for 75-100% they pay 40%. The overall cost for an entire month of sustained use comes to 70% of on-demand (i.e., 30% discount).

Customized. Google also offers a customized VM option, which enables users to purchase a VM with a custom cost based on a configurable number of cores and memory. Customized VMs can be used in conjunction with any of the purchasing options above, including the sustained-use discount. Customized VMs have the potential for significant cost savings by better matching job resource requirements to VM resources, thereby reducing wasted resources. However, this savings comes at an increased cost, which is currently 105% the normalized cost per core and GB-memory of an on-demand VM with pre-defined cores and memory allotment.

Spot Block. Amazon offers spot block VMs that have a short pre-defined lifetime of 1, 2, 3, 4, 5, or 6 hours. Spot block VMs are always revoked after their pre-defined lifetime (but typically not before), although users can terminate them early and only pay for the time they held them. Thus, spot block VMs have a maximum lifetime, but no time commitment. Spot block VMs cost 50-70% of the on-demand cost with higher discounts for shorter lifetimes. Spot block VMs are a form of *short-term reservation* that ensures the cloud platform is able to reclaim resources in the near future. Their

average discount is less than spot and near that of reserved. However, spot block VMs do not require a long time commitment and are designed for short tasks (<6 hours) that either have a deadline or cannot gracefully handle revocations, which makes them unsuitable for transient/spot VMs.

Scheduled Reserved. Amazon also offers a scheduled reserve option designed for workloads that do not run continuously but do run on a regular schedule, such as nightly batch jobs or financial simulations that run after the stock market closes each weekday afternoon. Scheduled reserved VMs enable users to define repeating daily, weekly, or monthly reservation schedules at hourly resolutions. For example, users could define a daily schedule that reserves a VM from 9 pm-12 am each day. As with the reserved option, scheduled reserved capacity is guaranteed and not revocable. However, the discount is much smaller, only 10% during off-peak weekend hours and 5% during peak weekday hours. Scheduled reserved are only offered for a 1-year term, and require users to purchase a schedule with a minimum of 1200 hours over the year. This option is also currently available in only 3 of the larger regions (Northern Virginia, Oregon, and Ireland).

2.2.1 Differences between Cloud Providers

As Table 2.1 shows, the major cloud providers offer slightly different VM purchasing options. Microsoft offers the simplest set with on-demand, 1- and 3-year reserved, and transient options. Google then adds the sustained-use discount for on-demand VMs, along with the ability to configure customized VMs with any purchasing option. In contrast, Amazon adds the spot block and scheduled reserved options. Note that Table 2.1's relative cost is an estimate across all the cloud providers. In general, Microsoft quotes similar prices (in the same way) as Amazon for the same purchasing options and VM types, while Google quotes prices slightly differently. However, the discounts offered for the purchasing options (even for comparable VM sizes) are not

directly comparable across cloud providers, since aspects of their infrastructure, such as the network and I/O bandwidth, and the resources may differ.

2.3 Workload Characteristics

In this work, we broadly characterize a job as either a highly available interactive service or a batch job. In general, interactive services are treated as foreground jobs, while the batch jobs run in the background, giving the interactive service priority over batch jobs.

Interactive services require very high service uptime i.e., little to no downtime. Examples of interactive services include nginx webserver, databases, etc. These services generally support front-end services like mobile applications, web applications, etc where they must respond to user requests in real-time at millisecond-scale latencies. Because of the performance requirements of interactive services, on-demand and reserved VMs are generally preferred for hosting (or running) them.

Batch Jobs are delay tolerant and stateful applications that typically use fault-tolerance to mitigate any failures. Examples include data-processing applications like Spark and distributed machine learning jobs like Tensorflow, PyTorch, etc. Resource requirements of batch jobs can range anywhere from "single" machine to "thousands" of machines as well. These batch job platforms like Spark and Tensorflow allow users to checkpoint the state and hence these jobs can handle failures by restarting from the previous state in the case. Since batch jobs can tolerate failures and delays, transient VMs are suitable for running batch jobs (that require a large number of computing resources) at a significantly lower cost.

2.4 Container Orchestration Platforms (COPs)

There are many publicly available COPs that offer similar functionality and support diverse workloads on large, mixed-use clusters, including Kubernetes [30], Mesos

(with Marathon) [43], and Docker Swarm [3]. These platforms not only manage the allocation of cluster resources between distributed applications encapsulated in containers but also provide a rich set of functions for supporting distributed applications and tightly integrate with IaaS cloud platforms. In addition, IaaS platforms now natively provide container orchestration platforms hosted on their VMs, such as Amazon's EC2 Container Service for Kubernetes (EKS) [2], Google's Kubernetes Engine (GKE) [6], and Azure Kubernetes Service (AKS) [1]. These native platforms are largely the same as the open-source platforms but are configured automatically by the cloud provider and support elastic autoscaling, i.e., by automatically allocating and releasing VMs as necessary.

A key assumption COPs make is that distributed applications that run on them can handle i) the failure or revocation of containers and ii) the allocation of new replacement containers. This assumption enables COPs i) to scale up to many thousands of servers, where some failures are inevitable, by automatically replacing failed containers with new containers and ii) to freely revoke containers from applications without affecting application correctness. COPs generally support two broad classes of applications—interactive services and batch jobs [81]. Interactive services must respond to user requests in real-time at millisecond-scale latencies, while batch jobs typically run in the background without strict deadlines. Internal support for revocations enables COPs to allocate available resources to batch jobs, while also enabling them to revoke these resources and re-allocate them to interactive services if their demand spikes.

Thus, distributed applications that run on COPs are designed to handle container failures, revocations, and re-allocations. Interactive services are typically designed to be stateless and often leverage load balancers natively built into COPs that spread requests among services' active containers. These load balancers automatically update the active set of containers as failures and revocations occur. In contrast, batch

jobs either restart from the beginning, or from the most recent checkpoint, on a revocation. Many distributed big data frameworks, such as Spark [91], Naiad [56], TensorFlow [13], etc., that cache large datasets in memory include built-in functions for checkpointing job state. Optimizing this checkpointing based on job characteristics and revocation rates is the subject of active research [63, 74, 89, 90].

CHAPTER 3

EXECUTING MIXED INTERACTIVE AND BATCH WORKLOADS ON TRANSIENT VMS

Increasingly, the resources under container orchestration platforms (COPs) like kubernetes are virtual machines (VMs) dynamically acquired from cloud platforms like Amazon AWS. COPs may choose from many different types of VMs offered by cloud platforms, which differ in their cost, performance, and availability. In particular, while *transient VMs* cost significantly less than on-demand VMs, platforms may revoke them at any time, causing them to become unavailable. While transient VMs' price is attractive, their unreliability is a problem for COPs designed to support mixed workloads composed of, not only batch tasks, but also long-lived interactive services with high availability requirements. In this chapter, we design TR-Kubernetes, a COP that optimizes the cost of executing mixed interactive and batch workloads on cloud platforms using transient VMs.

3.1 TR-Kubernetes Overview

TR-Kubernetes is a Container Orchestration Platform (COP) that uses unreliable transient cloud VMs to execute mixed batch/interactive workloads at low cost. To do so, TR-Kubernetes enables interactive services to explicitly specify their capacity availability requirement. For example, a distributed web service may specify that it needs the equivalent capacity of 500 `m4.large` Amazon EC2 VMs with 5 nines of availability (99.999%). TR-Kubernetes's provisioning policy then selects a mix of different transient VMs, from among the hundreds offered by cloud platforms, that satisfies the capacity availability requirement with high probability.

To enforce high availability using unreliable transient VMs, TR-Kubernetes must acquire *many more* transient VMs than necessary most of the time. Current transient VM prices are low enough that acquiring more transient VMs than necessary—in some cases many more—is still *much cheaper* than using on-demand VMs to satisfy the same capacity availability requirement. TR-Kubernetes automatically leverages this excess capacity to execute batch jobs at no additional cost. However, if interactive services ever require additional resources, due to increased demand or transient VM revocations, TR-Kubernetes leverages internal functions COPs already provide to revoke resources from the batch jobs and allocate them to the interactive services.

Interestingly, the reason COPs include internal support for revocations, and thus transience, is the same reason cloud platforms offer transient VMs: to increase overall cluster utilization. Existing integrations of COPs with cloud platforms typically run interactive services on high-cost on-demand VMs, and run batch jobs on low-cost transient VMs. In contrast, our hypothesis is that TR-Kubernetes can enable higher availability, better performance, and lower costs by running mixed interactive and batch workloads entirely on transient VMs.

3.2 TR-Kubernetes Design

In this work, we assume interactive services are stateless and leverage Kubernetes's built-in load balancer to distribute requests across VMs, it assumes any composition of VMs that satisfies the ECU requirement is acceptable, as the load balancer will distribute requests evenly based on each VM's resource capacity.

TR-Kubernetes's design relies heavily on existing functions built into Kubernetes, as well as other COPs. Figure 3.1 depicts TR-Kubernetes's extensions to Kubernetes. These include an offline tool that runs TR-Kubernetes's *provisioning algorithm* to generate service descriptions, which specify the transient VMs necessary to satisfy the capacity availability requirement. This service description is then submitted along

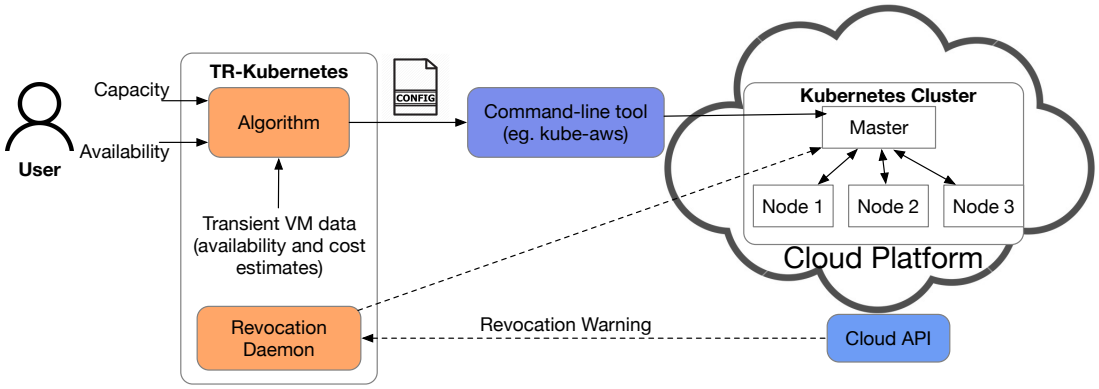


Figure 3.1: A depiction of TR-Kubernetes architecture.

with the job via the Kubernetes command-line tool. Since our prototype runs on EC2, users specify capacity in terms of EC2 Compute Units (ECUs), which is a relative measure of a VM's integer processing power [24]. As an example, an interactive service might specify that it requires 10,000 ECUs with an availability of 99.999%.

TR-Kubernetes's provisioning policy determines which transient VMs to request and how many based on their availability estimates, such that they satisfy the capacity availability requirement at the lowest cost. Once transient VMs are allocated based on a service description, TR-Kubernetes relies on functions already built into Kubernetes on AWS, an open source implementation of Kubernetes designed to run on VMs dynamically acquired from EC2. Kubernetes on AWS supports EC2 services, such as Auto Scaling and Spot Fleet, designed to automate resource allocation and revocation. As a result, Kubernetes on AWS can handle revocations by simply detecting them as failed VMs and removing them from the cluster.

TR-Kubernetes also employs an external *revocation daemon* that interacts with cloud APIs and Kubernetes to detect imminent revocations and proactively remove revoked VMs to minimize failed requests. In addition, if a revoked VM reduces the capacity of an interactive service below the requirement in its service description, Kubernetes's replication controller can automatically spawn replica container *pods*, i.e., co-located groups of containers, on the remaining VMs to replace the resources

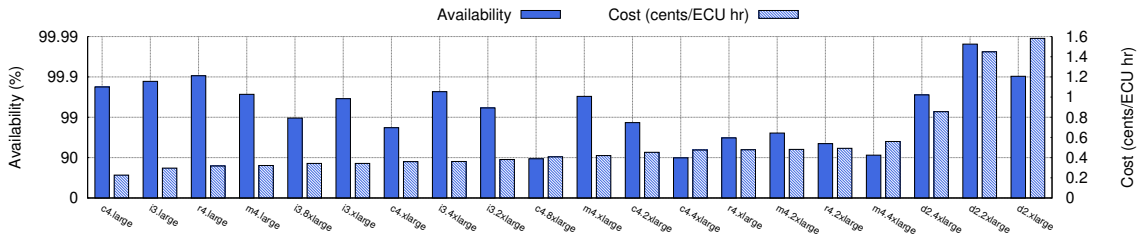


Figure 3.2: Comparison of average transient VM cost and availability in *us-west-1c* from 2017/9-12.

on the revoked VM. If necessary, to ensure the required capacity, Kubernetes will revoke resources internally from lower-priority batch jobs.

3.3 Provisioning Algorithm

The differences in cost between different combinations of on-demand VMs (with equivalent ECU or memory capacity) are small, since on-demand VMs (within each VM class) are consistently priced in proportion to their ECU and memory capacity. These cost differences, however, are often much higher for transient VMs at any given time. Figure 3.2 illustrates the complexity of selecting transient spot VMs in EC2 that are both low cost and satisfy a target availability. The figure shows that cost and availability are not correlated, such that the VM with the lowest price is not necessarily the most available, and that there are a wide range of options with different tradeoffs. Note that availability is on a log scale to emphasize that small differences are important.

Our provisioning policy addresses this problem by selecting transient VMs to jointly optimize both cost and availability subject to the availability target. To do so, TR-Kubernetes maintains a table of price and availability estimates for each transient VM. Given the table, computing the aggregate availability of different capacities for a pool of transient VMs is non-trivial, especially if transient VM availability is highly correlated. Fortunately, Figure 3.3 shows that availability for transient VMs is not highly correlated in EC2. While we show only 11 VM types from `us-west-1c` here,

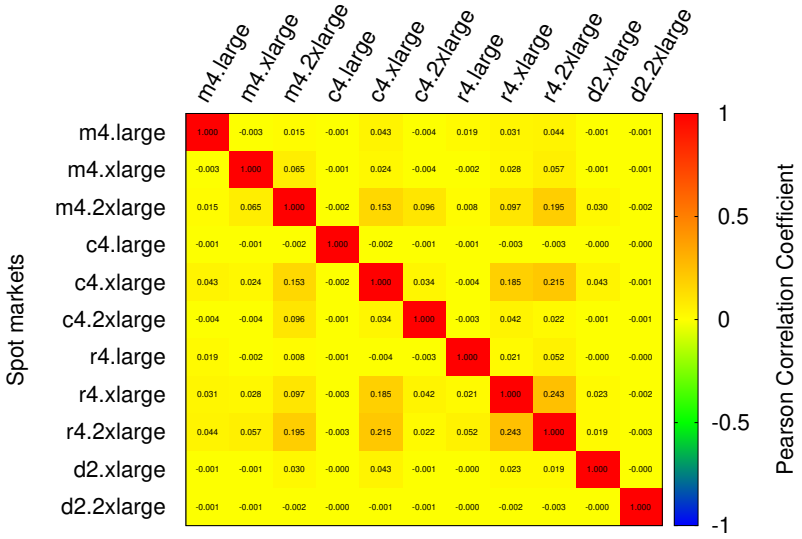


Figure 3.3: Heatmap showing the correlation in availability periods of spot VMs in `us-west-1c` from September to November 2017. The figure shows that availability periods are largely independent across spot VMs.

our analysis across the other regions, zones, and VM types yields similar results. Given this, in our analysis below, we assume availability periods are independent. Also note that transient VMs of the same type are perfectly correlated, since a single price dictates their availability *en masse*.

3.3.1 Computing the Availability of a Target Capacity

Before describing our provisioning algorithm, we first outline how to compute the availability of a target capacity C across a pool of transient VMs with different availabilities. Since transient VMs of a given type are either available or unavailable, our approach, described below, essentially computes the probability of all possible available/unavailable combinations, and then sums the probabilities of all combinations that yield a capacity $\geq C$. To do so, we first denote each transient VM's availability as p_i and its capacity as c_i. We can then represent a transient VM i as a polynomial $(Q_i(x))$ of degree $n_i \times c_i$, where we have n_i for each transient VM i.

$$Q_i(x) = (1 - p_i)x^0 + p_i x^{n_i c_i} \tag{3.1}$$

Here, the exponents of x represent the capacity of transient VMs of type i, while their coefficients represent the probability that a certain capacity is available (either zero or $n_i \times c_i$). We use this polynomial to indirectly represent the probability mass function (PMF) of transient VMs of type i. Of course, any single VM type is either available or unavailable, which yields a simple form for $Q_i(x)$.

To compute availability of a capacity C for a pool of N different transient VM types, with n_i of each type, we derive our representation of the probability mass function of the transient VM pool by simply multiplying the polynomials of each transient VM, as they are independent. This gives the polynomial representation Q_{pool}(x) for the transient VM pool shown below.

$$Q_{pool}(x) = \prod_{i=1}^{N} Q_i(x) \tag{3.2}$$

Similar to above, $Q_{pool}(x)$ represents the probability mass function of the entire transient VM pool, where x's exponents represent the capacity for some set of transient VMs, and the coefficients represent the probability that this set of transient VMs are the only ones that are available. From this equation, we can compute the availability at a target capacity m ($m \leq C$) by simply adding the coefficients of x's exponents, where the respective exponent is $\geq m$, as these are the combinations of transient VMs that yield greater than the capacity requirement. Note that the methodology above is equivalent to taking the convolution of multiple independent binomial random variables, where we represent i) each transient VM type's availability as an independent binomial random variable A_i $\forall i \in [1, N]$ and ii) the transient VM pool as the sum of the respective transient VMs' binomial random variables.

3.3.2 Greedy Algorithm

We next outline how TR-Kubernetes selects transient VMs for the pool to minimize cost, while satisfying the target level of availability for the specified capacity. The problem is complex, since there are hundreds of transient VM types within each

cloud region (including each AZ), and we may select multiple instances of any one transient VM. As a result, there are an exponential number of possible pools that satisfy the capacity availability requirement.

Our problem appears similar to a multi-dimensional unbounded knapsack problem, where the VMs are akin to items, ECUs and availabilities are akin to weight dimensions, VM pools are akin to knapsacks, and costs are akin to item value. However, there are two primary differences that prevent applying common techniques, such as dynamic programming, to the problem. First, in our problem we do not know know the final number of ECUs (knapsack size) required for a given target availability and secondly, availability dimension in our problem is not strictly additive, since the increase in availability from adding a new transient VM to a pool depends on the other transient VMs already in the pool.

Thus, we instead initially employed a simply greedy algorithm that selected transient VMs in order of their lowest cost (per ECU-hour) per marginal increase in availability of the specified capacity relative to the currently selected pool of transient VMs. If the currently selected pool has less than the target capacity, then we compute the marginal increase in availability of the capacity of the currently selected group. This approach favors low cost transient VMs that yield high availabilities. Note that we select based on the marginal increase in availability with respect to the currently selected group, and not the absolute availability of the transient VM.

When we directly employed the greedy approach above, we found that it did not work well because it applied the same weight to cost and availability. However, transient VM availability in cloud platforms is currently high (above 90%) and users typically reason about and specify availability based on small orders of magnitude, i.e., some number of 9s of availability. In contrast, users reason about and specify cost based on much larger orders of magnitude, i.e., reducing cost by 50% is significant, while reducing it by 1% or 0.1% is generally not. As a result, we modified our initial

approach above by quantifying availability on a logarithmic scale using the equation below, assuming transient VM availability is <100%, where p_i is the availability of transient VM's of type i and σ_i is its cost per ECU-hour.

$$\log(\frac{1}{p_i}) \times \sigma_i \tag{3.3}$$

Our algorithm greedily selects transient VMs based on the criteria above one by one until the pool satisfies the specified capacity availability requirement, or its cost exceeds the cost of using on-demand VMs to satisfy the requirement, in which case TR-Kubernetes requests on-demand VMs.

3.3.3 Supporting Multi-Tier Services

In Kubernetes, users specify multi-tier services by submitting a separate service description for each tier that specifies its resource requirements. Since a key goal of TR-Kubernetes is to minimally alter Kubernetes, we adopt the same approach with users independently specifying the capacity availability requirement of each tier. Of course, a key problem with such independent resource specifications is they do not allow users to specify an availability requirement for the aggregate multi-tier service. However, we can derive this availability from the requirements of the tiers.

We consider a multi-tier service to be available if the capacity requirement of each of its tiers is concurrently satisfied. If the availability of the transient VMs allocated in each tier is independent *and* the capacities are the same, then the availability of the aggregate service is simply $(p_i)^k \forall i \in [1, k]$, where p_i is the availability of the ith of k tiers. In this case, to enable an availability of p for the multi-tier service, users should specify an availability of $\sqrt[k]{p}$ for each tier. Thus, users must specify a higher availability for each tier relative to their desired availability for the multi-tier service. If each tier's availability is independent, but the capacity is different, then we can simply treat each tier as a single transient VM equal to the specified capacity with

availability p_i, and use the same approach as in 3.3.1 to compute the availability of the multi-tier service.

However, note that even when we determine each tier's transient VM allocation independently based on the greedy algorithm above, the resulting allocation is entirely dependent across tiers. That is, if each tier's capacity and availability requirement are the same, then *each tier's allocation will be exactly the same*, as our algorithm is deterministic. This also holds if the capacity of each tier is different, but their availability requirement is the same, since any tier's transient VM allocation will be a subset of the allocation for all tiers with a higher capacity. If both the capacity and availability of each tier are different, then the availability of the multi-tier service is bounded by the tier with the lowest availability.

3.4 Implementation

We implement TR-Kubernetes by minimally extending Kubernetes on AWS [4].

Prototype. By design, TR-Kubernetes minimally extends Kubernetes on AWS, as it is designed to be simple and exploit much of Kubernetes's (and other COPs') rich set of existing functions. The prototype uses Python bindings with the Boto3, numpy, and PyYAML python libraries. Kubernetes on AWS already enables users to create, update, and destroy Kubernetes clusters dynamically on AWS, either programmatically or using a command-line tool. While Kubernetes includes most of the functions TR-Kubernetes requires, they must be correctly configured. To create a cluster, the software first generates a template configuration file, where the user has to fill in certain cluster parameters, such as the EC2 key pair, DNS name, type of VMs to use, and the number in each node pool. Kubernetes on AWS acquires the cluster from EC2 based on this configuration file.

3.5 Evaluation

We evaluate TR-Kubernetes at small scale on EC2 using our prototype, and at large scale over a long period using publicly-available spot price traces. Our prototype results focus on quantifying the effect of revocations on an interactive services's performance and reliability. We also use spot price traces to quantify the ability of TR-Kubernetes to satisfy different capacity and availability requirements, and the resulting cost compared to using on-demand VMs.

We run all simulation experiments using spot price data from all 14 EC2 AZs in the U.S., and report error bars that represent the minimum and maximum of each metric. Our spot price data covers 3 months from September to December 2017, which is after the latest change in EC2's spot price algorithm. We consider all spot VM types except for GPUs and FPGAs, which are not general-purpose and require support from applications. We also only consider the latest (4th) generation of VM types, as of September 2017.

3.5.1 Prototype Results

Our prototype results focus on the application performance and reliability impact of revocations. Since substantial prior work has optimized batch jobs for revocations, e.g., by tuning fault tolerance mechanisms [39,42,64,74,89,90], our evaluation focuses on interactive service performance. Note that batch jobs run on TR-Kubernetes can leverage this prior work. For these experiments, we use a distributed web server that serves static content as a representative application. The server uses Kubernetes's built-in load balancer to distribute requests across 10 server replicas (running Nginx) hosted on `t2-medium` VMs in EC2. We omit results from experiments using multiple tiers that yielded the same result due to space limitations.

Throughput. We first examine application performance under increasing revocation frequencies. Since replacements for revoked VMs must re-warm their cache, frequent

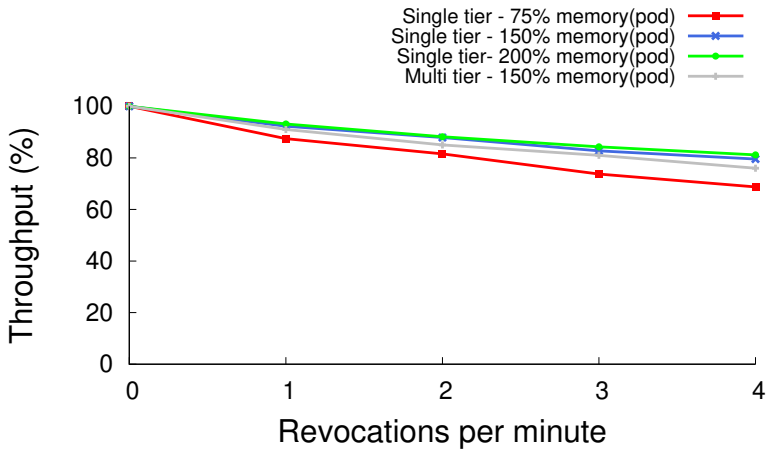


Figure 3.4: Distributed web server throughput as a function of revocation frequency for different working set sizes.

revocations have the potential to degrade server throughput and latency. In this case, we seed our replicas with a number of image files, and use the ApacheBench (ab) web server benchmarking tool by configuring 4 clients to send 2000 request over a 8 minute period. We vary the size of the static content, i.e., image files, in each experiment such that either i) all the content fits in the VM's memory, ii) only some of the content fits in the VM's memory, requiring some content to be served from disk, or iii) most of the content resides on and must be accessed from disk. We then vary the revocation frequency from 0 revocations/minute to as high as 4 revocations.

Figure 3.4 presents the results, which show that the larger the size of the static content relative to memory, the lower the effect of revocations on throughput. When the content fits in memory, all content is served from the memory cache, such that when a revocation occurs and flushes this cache, there is a significant decrease in throughput as the cache re-warms. This effect is less pronounced as the size of the content increases, as larger fractions of requests must be served from disk instead of the cache. Even in the worst case, significant throughput degradation requires much higher revocation frequencies. Of course, dynamic content would not be subject to these caching penalties due to revocations. Thus, the performance impact of even high revocation frequencies on interactive server-based applications is likely to below.

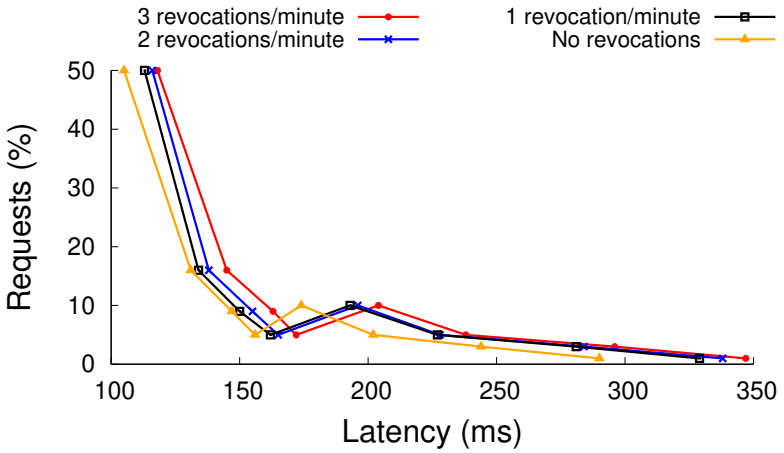


Figure 3.5: Latency distribution of a distributed web server for different revocation rates.

Latency. We next examine the effect of revocations on service latency. As noted above, frequent revocations result in performance degradation. To illustrate, we use the same distributed web server configuration as above, and plot latency distribution for the requests in Figure 3.5. The graph shows that increases in the revocation rate slightly increase the latency by shifting the distribution to the right. Even so, the difference between no revocations, and 1 revocation/minute, which is an extremely high revocation rate, is small. Since revocation rates this high are unlikely in practice, revocations are unlikely to have a significant effect on average or tail latency.

Reliability. For interactive services with high throughput, simply treating VM revocations as failures can cause outstanding requests on the VM to fail. As discussed in §3.2, TR-Kubernetes's revocation daemon uses an advance warning of the revocation to gracefully coordinate VM removal to minimize the number of failed requests. To demonstrate, we use the same distributed web server setup as above with 10 server replicas, but serving out a simple static html page. We again use ApacheBench (ab) to benchmark the server by configuring 20 clients to send 450,000 request over a 5 minute period. We then vary the rate of VM revocation, and measure the number of failed requests both with and without TR-Kubernetes's revocation daemon.

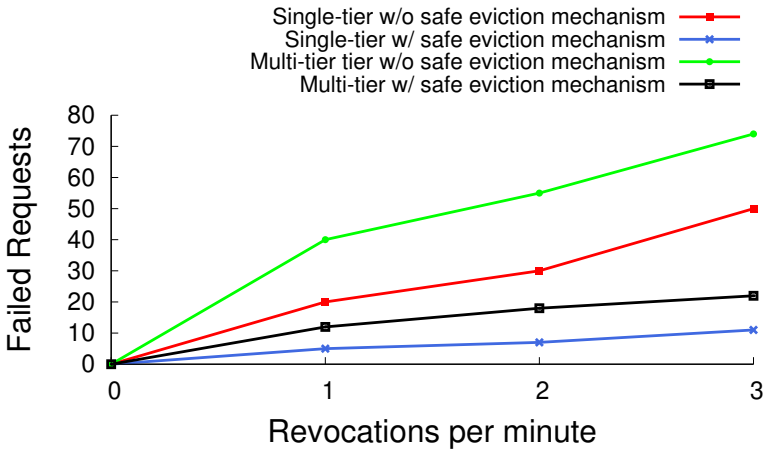


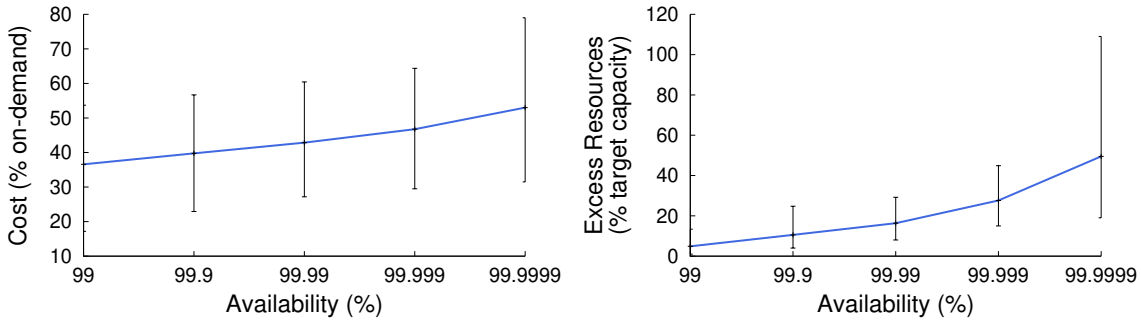
Figure 3.6: Failed server requests (HTTP 200) as a function of revocation rate with and without the revocation daemon.

Figure 3.6 shows the results. We see that when vanilla Kubernetes treats revocations as failures, the number of failed requests increases linearly with the revocation rate. In contrast, TR-Kubernetes's revocation daemon decreases the number of failed requests by >5×. These few failed requests occur when the revocation daemon interacts with the Kubernetes master to make the VM un-schedulable, as this action is likely not atomic and must coordinate with both the Kubernetes scheduler and the load balancer. Hence, the number of failed requests is small compared with the overall number of requests, even at high revocation rates.

3.5.2 Cost and Availability Analysis

We next analyze the potential cost and availability of an interactive service using TR-Kubernetes over 3 months using traces of EC2 spot prices to infer realistic cost and availability characteristics. Our cost analysis also includes the excess resources an interactive service must acquire to maintain its capacity availability requirement.

Cost Analysis. Figure 3.7 quantifies both the cost (left) and excess resources (right) on the y-axis for different availability targets on the x-axis for a capacity requirement of 5000 ECUs. We normalize our cost to the cost of satisfying the target 5000 ECU capacity using the on-demand VM with the lowest cost per ECU-hour. The (left) figure shows that, as expected, the cost increases as we increase the availability target



(a) Cost (b) Excess Resources

Figure 3.7: The cost relative to on-demand (a) and excess resources relative to the target capacity (b) for a target capacity of 5000 ECUs as availability varies.

(left), although the increase in cost is not substantial when going from 2 nines availability (at 38% the on-demand price) to 6 nines availability (at 53% of the on-demand price). While there is some variance across AZs, as indicated by the error bars, in all cases, the cost for even 6 nines availability, which translates to 31.5 seconds of capacity below the requirement per year, is less than the cost of the equivalent on-demand VMs.

Figure 3.7(right) then shows the amount of excess unreliable transient VM capacity we must purchase to enforce our availability requirement. As expected, the higher the availability requirement, the more excess unreliable capacity we must purchase to enforce it. This excess capacity is the reason for the cost increase in Figure 3.7(right). At 5 nines availability, we must purchase 30% more capacity than necessary, and for 6 nines availability, we must purchase 50% more capacity. As discussed below, TR-Kubernetes leverages this excess capacity to execute low-priority batch jobs.

To Summarize, Figure 3.7(left) shows that TR-Kubernetes can achieve *higher availabilities* at a *lower cost*, ranging from 20% to 80% of the on-demand price depending on the availability requirement.

Availability Analysis.

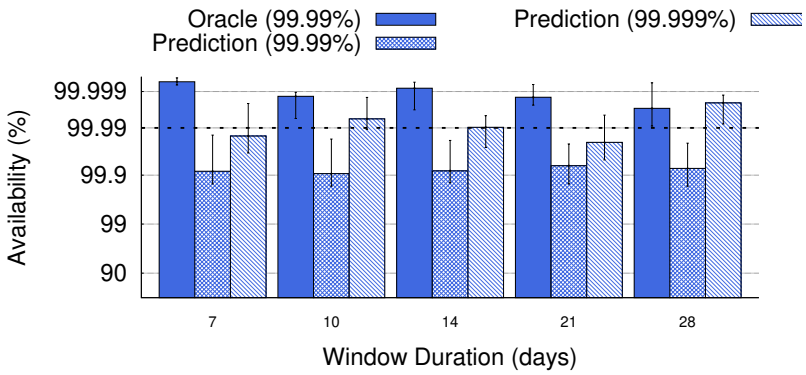


Figure 3.8: Unpredictable prices affect TR-Kubernetes's accuracy at satisfying its availability target.

The analysis above uses the actual availability of the transient VMs based on their price data over the 3 month trace period. As a result, our provisioning algorithm always satisfies the capacity availability requirement. However, in practice, spot prices and availability are not known *a priori*, requiring us to first estimate spot prices and availability based on recent history. To understand how these estimates affect our accuracy at satisfying the availability target, we experimented with different prediction window durations in Figure 3.8. The window specifies the interval length over which we estimate VM availability, such that a window of seven days estimates availability over the previous seven days. We again use a capacity of 5000 ECUs and an availability target of 99.99%. The graph then compares the realized availability based on the predicted estimates with the target availability.

Figure 3.8 shows that the realized availability across prediction windows remains relatively stable, but tends to be one 9 less than the target. The error derives from changes in VM prices and availability over time. In contrast, the oracle that assumes future knowledge of availability and prices always yields an availability strictly greater than the target (since it continues adding VMs until it reaches the target). One way to account for this error is to leverage prior work on spot price prediction to attempt to better predict future prices, rather than use simple historical estimates. Another

way to address this problem is to simply set the target higher than required, which we show in the figure by also including the availability for a target for 5 nines.

3.6 Related Works

There has been a significant amount of prior work focuses on optimizing the selection of transient VMs and configuring of fault-tolerance mechanisms, such as checkpointing, replication, and migration, to minimize the performance impact of revocations [49,63–65,74,87,90,95]. In general, TR-Kubernetes differs from this work in its focus on supporting the availability requirements of interactive workloads on transient VMs, and using the excess resources from doing so to execute batch jobs.

There has also been some research that focuses on running interactive workloads on transient VMs. For example, SpotCheck [65] operates at the virtualization layer and maintains a synchronized in-memory backup of the transient VM memory state, such that if a revocation occurs, the backup VM can seamlessly take over [69], similar to VM-based primary-backup systems, such as Remus [32]. SpotCheck is complex to implement, requiring the use of nested virtualization and the propagation of individual memory writes to the backup server. By contrast, TR-Kubernetes is simple, and leverages much of the functionality already present in COPs, which makes it more likely to be adopted in practice. However, unlike SpotCheck, TR-Kubernetes only supports stateless interactive services using Kubernetes' built-in load balancer.

Similarly, Tributary [5] applies an approach called "spot dancing," which leverages spot VMs to support latency SLOs for interactive services at low cost, by intentionally selecting and bidding on spot VMs to cause revocations. Tributary does have similar dynamics as TR-Kubernetes in that it provisions many more resources than necessary, which provides headroom for interactive services to absorb demand spikes. However, Tributary does not ensure availability requirements or leverage additional resources to execute batch jobs. Tributary also does not build on existing COPs to simplify its

use or adoption. More generally, our approach is not specific to spot VMs or EC2's billing model, as we only use the spot price traces to derive availability periods. As a result, TR-Kubernetes is applicable to any variant of transient VMs with such a characterization of availability.

3.7 Conclusion and Status

In this chapter, we presented TR-Kubernetes, a minimal extension of Kubernetes that executes mixed interactive and batch workloads on unreliable transient VMs dynamically acquired from cloud platforms. Unlike prior work, TR-Kubernetes is practical to implement, as it primarily requires selecting the right set of transient VMs based on their expected cost, availability, and capacity. We implement our approach on EC2 and show that, when compared to running interactive services on on-demand VMs, TR-Kubernetes is capable of lowering costs (by 53%) and providing higher availability (99.999%).

Status. TR-Kubernetes have been evaluated on a 3-month EC2 trace via simulation and small scale real-world experiments as well. Additional details on its design, implementation, and evaluation are in [21].

CHAPTER 4

UNDERSTANDING SYNCHRONIZATION COSTS FOR DISTRIBUTED ML ON TRANSIENT VMS

Cloud platforms often execute parallel batch applications, such as distributed machine learning (ML), that include numerous synchronization barriers. Unfortunately, these barriers significantly degrade application performance by reducing it to that of the slowest "straggler" task. To address the problem, researchers have proposed numerous straggler mitigation techniques, including speculatively re-executing straggler tasks and various relaxations of strict barrier semantics. While these techniques improve parallel application performance, they incur a cost in terms of the resources wasted re-executing tasks or waiting. While transient cloud resources' low list price is attractive, revocations increase the frequency and severity of stragglers, which decreases parallel job performance and increases overall execution cost. In this chapter, we develop simple analytical models of different straggler mitigation techniques to better understand the cost of synchronization and compare their cost and performance on on-demand and transient resources.

4.1 Motivation

Public cloud platforms provide users access to an essentially unlimited number of servers on demand without requiring a large capital investment. Thus, enterprises are increasingly leveraging public clouds to run large-scale workloads, often for distributed data processing, across hundreds-to-thousands of servers. Since distributed data processing platforms, including Hadoop [68], Spark [91], Tensorflow [13], PyTorch [61]

and parameter servers (PSs) [54], simplify running jobs across many resources, they have become the dominant platforms for leveraging cloud resources. Many general platforms, including Hadoop and Spark, adopt a bulk synchronous processing (BSP) model [79], which defines synchronization barriers that prevent any task from advancing beyond specified points until all tasks have reached those points.

Unfortunately, synchronization barriers significantly degrade parallel application performance by reducing it to that of the slowest "straggler" task. The original work on MapReduce identified such stragglers, which arise for many reasons [23, 27, 33, 62]. As a result, prior work has developed many techniques to identify and mitigate the effect of stragglers on performance. While Hadoop and Spark are general data processing platforms, recent frameworks, such as Tensorflow and PSs, focus specifically on distributed machine learning (ML) jobs, given their increasing importance. While these platforms must also address stragglers, distributed ML jobs enable new approaches beyond speculative execution [31, 44, 54].

Importantly, while stragglers significantly degrade application performance, as indicated above, they also significantly increase application *cost* when run in the cloud. Cloud platforms charge users for the time they use a server at fine-grained intervals, e.g., every second or minute. As a result, any time tasks spend waiting idle at barriers results in resource waste that translates into a higher cost. In general, prior work focuses on dedicated clusters in data centers and thus *does not* consider cloud platforms' fine-grained costs when evaluating their techniques.

Yet, cost, and *not performance*, is the dominant metric when using cloud platforms, as it is nearly always possible to increase performance by acquiring more cloud resources for an increased cost, although the relationship may not be linear. However, optimizing cost tends to differ from optimizing performance because cloud platforms offer servers under multiple different contract options. In particular, cloud platforms sell their excess capacity at highly discounted prices in the form of *transient*

servers [71], which they may revoke at any time to satisfy new requests for *on-demand servers*. While parallel batch jobs are ideal candidates for running on cheap transient servers, revocations degrade their usable computational capacity.

Currently, cloud platforms *do not* reveal any information about the revocation characteristics of transient servers [66]. Thus, transient servers essentially define a new type of cloud server that yields a probabilistic computational capacity, dictated by its unknown revocation characteristics, but where users pay a highly discounted fixed price per unit time. Despite the discounted price, using a transient server could result in a higher overall execution cost than using an on-demand server if it yields a low capacity (due to a high revocation rate), and thus takes significantly longer to complete a job.

For parallel jobs, transient servers also increase the likelihood of stragglers compared to prior work, as any revocation results in straggling, which increases resource waste and further increases overall cost compared to using on-demand servers. While straggler mitigation techniques can reduce the cost of stragglers, they each impose their own cost, making it unclear which technique is optimal and whether any provide a net cost benefit. To better understand the design space, we develop simple analytical models to quantify and compare the expected performance and cost of executing parallel jobs using different straggler mitigation techniques on both on-demand and transient cloud resources.

4.2 Model Overview

We first provide an overview of our basic model, and then discuss representative baseline values for its parameters.

Name	Parameter	Units	Range
Workload	W	#Operations	$W \geq 0$
Performance	s	#Operations/time	$0 < s \leq 1$
Price	p	$/time	$c > 0$
Parallelism	k	#	$k > 0$
Barriers	b	#	$b \geq 0$
Network Overhead	n	time	$n \geq 0$
Discount Factor	f	%	$0 \leq f < 1$
Backup Replicas	r	#	$r \geq 0$
Staleness Parameter	d	#	$1 \leq p \leq b$
Drop Parameter	N	#	$N \geq 0$
Total Time	T	time	$T \geq 0$
Total Cost	C	$	$C \geq 0$

Table 4.1: Name and description of our model's parameters, including their units and range.

4.2.1 Basic Model

Table 4.1 shows the name and description of our model's parameters, including their measurement units and range.

We assume parallel jobs must complete some workload W that requires executing some number of abstract operations, which represent a collection of CPU instructions and I/O operations. Servers execute these operations at a rate s, in operations per unit time, based on their performance capacity. We normalize s relative to the capacity of an on-demand server of a specific type, that experiences no revocations. Thus, on-demand servers of the specified type complete operations at a normalized rate of $s = 1$ operation per unit time. For transient cloud servers, we model s as a random variable, since revocation rates are not revealed by cloud platforms and are thus opaque to users. Since realistic revocation characteristics are unknown, and for simplicity, our model assumes transient servers yield a uniformly random performance s between 0 and 1 with an average performance of $s = 0.5$.

As with today's cloud platforms, our model assumes that on-demand servers with $s = 1$ incur a fixed-price p in dollars per unit time, while transient servers with $0 < s < 1$ are discounted by a factor $0 < f < 1$, which results in a fixed-price of

Name	Parameter	Baseline Value
Workload	W	8000
Parallelism	k	8
Barriers	b	500
Network Overhead	n	0.175
Discount Factor	f	0.9

Table 4.2: Representative model parameter values for baseline job.

$(1 - f) \times p$. Public cloud platforms currently discount transient servers up to 90%, or $f = 0.9$. However, note that since we assume transient servers yield an average performance of only $s = 0.5$, the discount in the expected cost C to complete a job, when accounting for the performance overhead of revocations, is actually only $0.5 \times 0.9 = 45\%$, as the jobs must run for longer compared to on-demand servers.

We adopt a simple model for parallel jobs: their total workload W is initially divided evenly across b barrier intervals and k parallel tasks running on separate (on-demand or transient) servers. Our model also assumes every barrier incurs some network overhead to communicate its results to other tasks. We model this overhead per barrier as being linear in the number of parallel tasks, as suggested in prior work [44], resulting in a total delay across all barriers of $n \times b \times k$, where n is a constant representing network overhead. Given our model above, we compute a parallel job's expected running time T and cost C to execute a workload W. The running time is simply the sum of the time for computing between barriers, including the time any servers spend waiting, and for communicating at barriers. Thus, using our notation from Table 4.1, we derive the expected running time as below.

$$T = \frac{W}{s \times k} + (n \times b \times k) \tag{4.1}$$

$$C = (1 - f) \times p \times k \times T \tag{4.2}$$

4.2.2 Representative Baseline Parameter Values

Our model above includes many parameters that affect a parallel job's completion time and cost. Rather than explore the entire parameter space, we define representative values for these parameters to serve as a baseline for comparison. Table 4.2 shows these representative values. We extract values for workload (W), degree of parallelism (k), number of barriers (b), and network overhead (n) from experiments performed in recent work on stale synchronization for distributed ML [44]. This representative parallel job took ∼1700 seconds to complete using 8 parallel tasks ($k = 8$), 500 barriers ($b = 500$), and a network overhead constant of 0.175 seconds ($n = 0.175$). This results in a workload W of 8000 in our model when run on homogeneous on-demand servers using BSP, assuming no stragglers.

As with any model, ours is not perfect and does not capture many job and resource characteristics that impact performance and cost. For example, unlike prior work, we only model stragglers caused by transient server revocations, and not other reasons. We also do not model the effect of different synchronization approaches on algorithmic running time and correctness. As a result, we intend our analysis to only highlight trends in job performance and cost for different synchronization approaches as the parameters change with a focus on scalability, i.e., an increasing degree of parallelism (k). Finally, we *do not* intend our model to be predictive, and thus, in practice, the precise performance and cost for even our baseline job may differ from our model's estimate.

4.3 Comparing Synchronization Models

Given the model from the previous section, we derive the expected running time T and cost C to execute a parallel job on on-demand and transient servers using different synchronization models and straggler mitigation techniques.

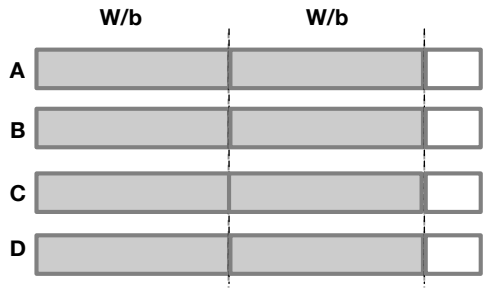


Figure 4.1: Parallel job using BSP on on-demand servers.

4.3.1 BSP on On-demand Servers

The simplest case is to use bulk synchronous processing (BSP) on on-demand servers, as done by Hadoop, Spark, and other distributed data processing platforms. Since, in our model, on-demand servers experience no revocations, they exhibit no stragglers, and thus we do not employ any straggler mitigation technique in this case. Figure 4.1 depicts a parallel job using BSP on homogeneous on-demand servers, where each horizontal progress bar is a task running across time on a different server, and the vertical dotted lines represent barriers. The $\frac{W}{b}$ terms above the progress bars represent the expected work done by all tasks over each barrier interval. We simplify Equation 4.1 and Equation 4.2 by setting $s = 1$ and $f = 0$ for on-demand servers, yielding an expected[1] running time using BSP on on-demand servers as follows.

$$T = \frac{W}{k} + (n \times b \times k) \tag{4.3}$$

$$C = p \times k \times T \tag{4.4}$$

Figure 4.2 then plots the speedup (left y-axis) and cost (right y-axis) of executing our representative parallel job from Section 4.2.2 as k increases. Here, the speedup and cost is normalized relative to their values when running the job on a single on-demand server with no barriers, i.e., $s = 1$, $k = 1$, and $b = 0$. The graph shows that as we increase k, the speedup, as with any parallel job, increases up to a point where

[1]In this case, s is actually deterministic.

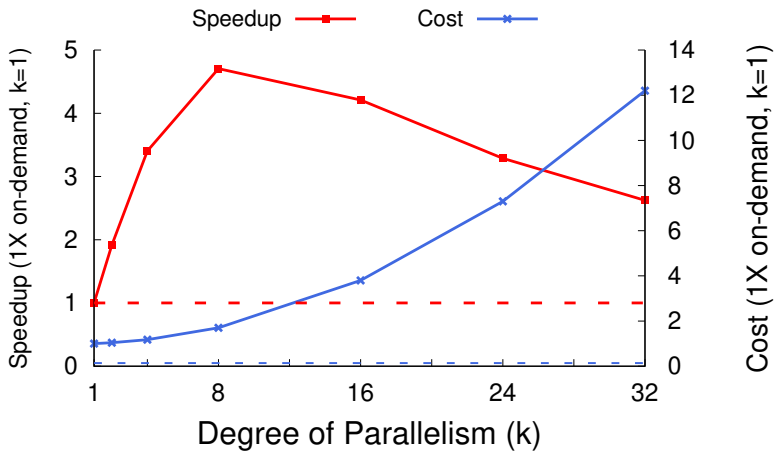


Figure 4.2: Speedup and cost of executing our representative parallel job using BSP on on-demand cloud servers as the degree of parallelism increases.

the communication delay at each barrier begins to offset the benefit of using more resources. In contrast, the overall execution cost C rises dramatically as k increases, as the increasing communication delays at barriers result in wasted time where the parallel job is paying for computing capacity but not using it.

Result: *Increasing the degree of parallelism k when using on-demand servers with BSP improves performance up to a point, but at an increasingly high cost that scales super-linearly.*

4.3.2 BSP on Transient Servers

Based on the high cost above of using BSP with on-demand servers, transient servers are a potentially attractive option for executing large-scale parallel batch jobs due to their low price. However, as discussed in Section 4.2.1, revocations decrease their usable performance capacity s, which we model as being uniformly random in the range $[0, 1]$. Importantly, recall that under BSP the slowest straggler task to reach a barrier dictates when all other (faster) tasks can proceed past the barrier. Thus, while cheaper, transient servers cause stragglers that reduce performance by increasing waiting time and resource waste. We must account for this waste and transient servers' discount when computing their expected running time and cost.

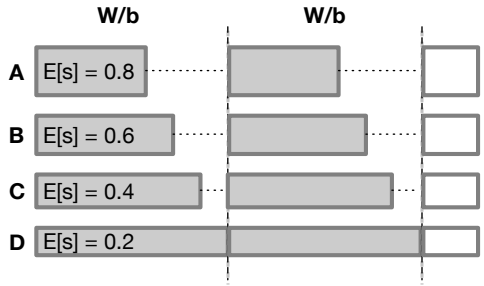


Figure 4.3: Parallel job using BSP on transient servers.

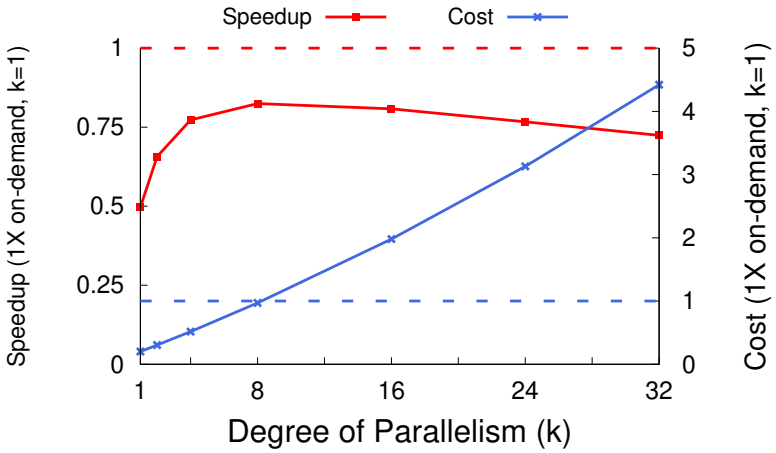


Figure 4.4: Speedup and cost of executing our representative parallel job using BSP on transient cloud servers as the degree of parallelism increases.

To do so, we must determine the expected speed of the slowest server: since we assume transient server performance s is uniformly distributed, this is equivalent to finding the expected minimum value when drawing k uniformly random numbers in the range $[0, 1]$. As a result, the expected performance for the fastest of k servers should be $s = \frac{k}{1+k}$ and for the slowest server should be $s = \frac{1}{1+k}$. Since the expected performance of *all* transient servers is dictated by the performance of the slowest expected server, we can reduce Equation 4.1 by simply substituting s with $\frac{1}{1+k}$. The cost C remains the same as in Equation 4.2.

$$T = \frac{W(1+k)}{k} + (n \times b \times k) \tag{4.5}$$

Figure 4.3 illustrates the expected performance s for each transient server when $k = 4$ under our model. Here, the progress bar's width represents each task's expected

performance s,[2] and the area of the progress bar represents total work, such that within each barrier interval, the area of each progress bar is equal.

Similar to Figure 4.2, Figure 4.4 plots the speedup (left y-axis) and cost (right y-axis) of executing our representative parallel job on transient servers as k increases. The graph shows that a single transient server incurs an expected speedup of 0.5× (or equivalently a slowdown of 2×), since it runs at half the expected speed of an on-demand server. As k increases, similar to above, expected performance increases up to a point where communication delays offset the benefit of adding more servers. However, based on Equation 4.5, as $k \to \infty$, the expected speedup with transient servers can never exceed that of using a single on-demand server with $s = 1$. However, despite their low performance, due to their 90% discount, transient servers offer a lower overall cost than using a single on-demand server for up to $k = 8$.

Result: *Increasing the degree of parallelism k when using transient servers with BSP improves cost up to a point (based on their discount), but incurs an increasingly high performance penalty due to their lower and non-uniform expected performance, which diminishes their cost advantage.*

4.3.3 BSP on Transient Servers with Backup Replica Tasks

Prior work proposes handling stragglers by identifying them, submitting a backup replica task for them, and then accepting the result of whichever task finishes first (and cancelling the other task) [23, 33, 92]. We model this approach by assuming that we can always immediately identify the slowest server(s) and submit backup replica tasks for them. While this assumption is not realistic, since we cannot assess transient server performance due to their unknown revocation rates, it serves as an upper bound on the performance and cost advantage of using backup tasks.

[2]In all figures, we denote performance using $E[s]$ to emphasize that transient server performance is an expected value, and not deterministic.

Figure 4.5: Parallel job using BSP on transient servers with backup replica tasks where $k = 4$ and $r = 1$.

Figure 4.5 illustrates a parallel job using BSP on transient servers with backup tasks, where our degree of parallelism $k = 4$ and the number of replicas $r = 1$. The red X represents that the replica task is cancelled once one of the replicas reaches the barrier. Note that the other replica is cancelled in the second barrier interval to illustrate that performance of transient servers is probabilistic and can change over time. As shown, the replica effectively increases the expected speed of the slowest task from $s = 0.17$ to $s = 0.33$, by allowing us to discard the slowest task, but at an additional cost for the replica. Thus, the expected performance s of the slowest task is a function of both the degree of parallelism k and the number of replicas r, as shown below.

$$s = \frac{1 + r}{1 + k + r} \tag{4.6}$$

$$T = \frac{W(1 + r + k)}{k(1 + r)} + (n \times b \times k) \tag{4.7}$$

$$C = (1 - f) \times c \times T \times (k + r) \tag{4.8}$$

Figure 4.6 then plots the speedup (left) and cost (right) of executing our parallel job on transient servers with different numbers of replicas as k increases. Note that $r = 0$ represents using BSP on transient servers with no replicas and is equivalent to Figure 4.4's speedup and cost. The graph shows that spawning backup replica tasks

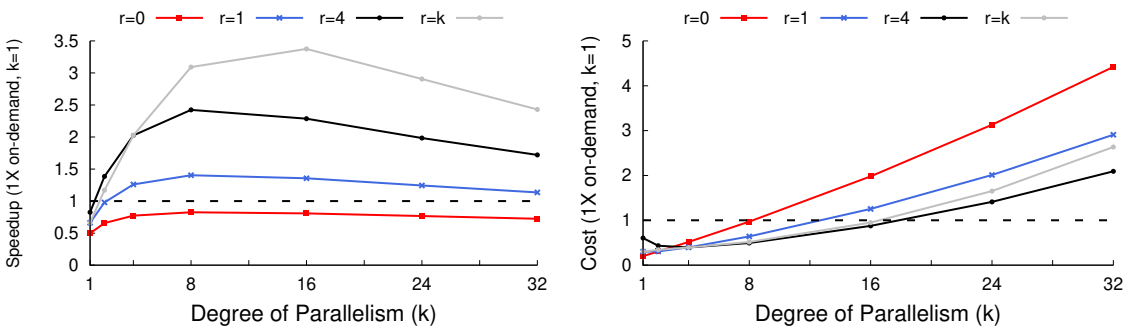


Figure 4.6: The speedup (left) and cost (right) of executing our representative parallel job with different numbers of backup replica tasks using BSP on transient cloud servers as the degree of parallelism increases.

improves both speedup and cost relative to not using replicas. In addition, unlike with no replicas, backup replica tasks enable the speedup to exceed 1×. However, replicas introduce some interesting tradeoffs. As expected, using more replicas always increases the speedup, as shown in the left figure, but the number of replicas that minimizes the cost is unclear, as $r = 4$ replicas is cheaper than both $r = 1$ replica and the extreme case of $r = k$ replicas. Overall, using replicas widens the values of k that yield both a speedup and cost advantage compared to not using replicas.

Result: *Increasing the degree of parallelism k when using transient servers with BSP and backup task replicas strictly improves the speedup and cost compared to not using replicas, and enables speedups greater than* 1.

4.3.4 Bounded Staleness on Transient Servers

Another approach for mitigating the impact of stragglers is to enable threads to perform a bounded amount of work past each barrier [31, 44, 54]. This approach reduces the time that fast tasks wait for slow ones, and can also reduce communication costs, as it effectively reduces the number of "real" barriers. The tradeoff is that, for distributed ML in particular, some tasks may access "stale" global parameter values that do not reflect all tasks' updates, which may impact a job's algorithmic convergence time and accuracy.

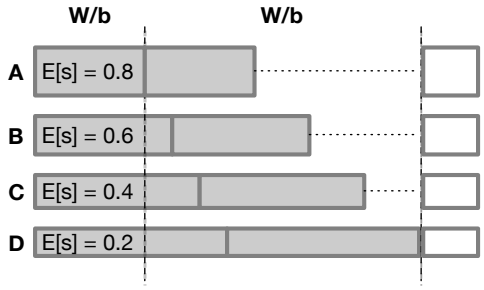


Figure 4.7: Parallel job using bounded staleness on transient servers.

We model this approach by simply introducing a staleness parameter d, similar to the one in [44], that reduces the number of barriers by a factor d. As a result, the expected running time T is the same as in Equation 4.5 but substituting $\frac{b}{d}$ for b, as shown below. The cost C is the same as Equation 4.2, but includes the T below.

$$T = \frac{W(1+k)}{k} + (n \times \frac{b}{d} \times k) \tag{4.9}$$

Figure 4.7 illustrates bounded staleness: typically, the barriers would be defined by when the slowest task, in this case D, completes its work, indicated by the vertical line in D's progress bar. However, with bounded staleness, the other tasks may perform a bounded amount of work past the barrier, which has the effect of making the effective barrier intervals longer.

Figure 4.8 shows the speedup (left) and cost (right) as the degree of parallelism changes under bounded staleness with different staleness parameters d. As expected, increasing the staleness parameter increases the speedup and decreases the cost by reducing the communication delays. However, even for the maximum value of $d = 500$, resulting in a single barrier, there is never a speedup compared to using a single on-demand server, and the cost becomes higher once $k > 8$. Bounded staleness is also worse in terms of both speedup and cost when compared to using backup replica tasks.

Bounded staleness is really only effective at mitigating the impact of stragglers that are rare and temporary. In these scenarios, the expected case is similar to using BSP with on-demand servers. Bounded staleness enables tasks to reduce (or

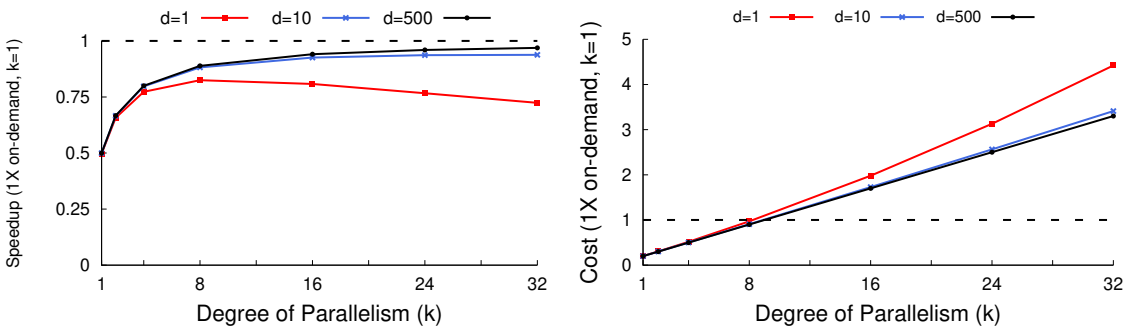


Figure 4.8: The speedup (top) and cost (bottom) of executing our representative parallel job with different staleness parameters d under bounded staleness on transient cloud servers as the degree of parallelism increases.

eliminate) any waiting and waste that might occur due to a few temporary stragglers. In contrast, when analyzing our model of transient servers, stragglers *are the expected case* and thus the fast tasks *always* must eventually wait for the stragglers after proceeding a bounded amount past a barrier. In this case, bounded staleness provides little benefit beyond reducing communication delays related to the number of barriers, which are not dominant at values of $k \leq 32$ in the graph.

Result: *Increasing the degree of parallelism k when using transient servers with BSP under bounded staleness is worse in terms of speedup and cost than using backup replica tasks.*

4.3.5 Partial Barriers on Transient Servers

The previous approaches extend the BSP model to mitigate stragglers. Prior work has also proposed "looser" synchronization models that relax the strict barrier semantics of BSP [15]. As one example, *partial barriers* mitigates stragglers by releasing the barrier once some number of tasks have reached it, and then cancels (or drops) the other tasks and re-distributes their work across all the servers. While prior work dynamically determines this release point based on the arrival rate of tasks to barriers, our model simply defines a drop parameter N, such that we release the barrier once $k - N$ tasks have reached it.

While such partial barriers are not applicable to all parallel jobs, they are applicable to distributed ML, as well as other examples cited in prior work [15]. Unlike with prior approaches, modeling partial barriers requires us to shift the cancelled work of slow tasks to the next barrier interval. Thus, the amount of work assigned to tasks per barrier interval increases as the job progresses. Our model redistributes this cancelled work equally across all servers in the next barrier interval. This results in work from slow tasks being shifted to faster servers.

We derive the expected running time T to complete a parallel job when using partial barriers as below.

$$T = \frac{W(1+k)}{b \times k \times (1+N)} \times [\sum_{i=2}^{b}[b-(i-1)](\frac{N}{k})^{i-2}] + \frac{W(1+k)}{b \times k} \times [\sum_{i=1}^{b}(\frac{N}{k})^{i-1}] + (n \times b \times k) \quad (4.10)$$

While we omit a full explanation due to space limitations, the first additive (top) term of this equation is similar to Equation 4.7 for the expected time when using backup replica tasks. Essentially, by dropping slow tasks, the overall speed becomes a function of the slowest non-dropped task, which is dictated by N (as opposed to r in Equation 4.7). However, unlike with backup replicas, we must account for the dropped work, which gets added to the work done next barrier interval and is equally distributed across all servers. The summation in the first term is the sum of the expected work that gets shifted to each barrier interval. The second additive term represents the expected running time required to finish the job after the final barrier, which is dictated by the speed of the slowest server, as we do not permit slow tasks to be dropped after the final barrier. This is why there is no N in the second additive term. The last term is the same communication delay as before.

The expected cost C is simply $(1-f) \times p \times T \times k$ as in the basic model, as this approach, unlike with replicas, uses no additional servers. Figure 4.9 illustrates this

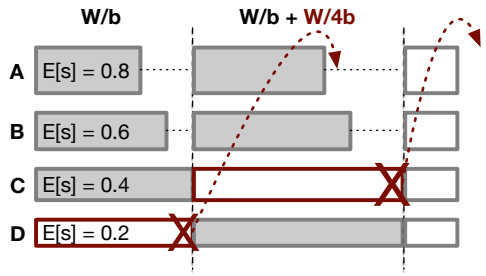


Figure 4.9: Parallel job using partial barriers on transient servers.

approach for $k = 4$, where the slowest task is dropped at the first barrier and its work is shifted to the next interval, as indicated by the red term added to the work that interval.

Figure 4.10 shows the speedup (left) and cost (right) of using partial barriers on transient servers for different values of N. N defines a tradeoff such that higher values increase the speed of the "slowest" server that reaches the barrier before it is released, but it requires cancelling and re-executing more work in the next barrier interval, which increases resource waste. In this case, the extreme points ($N = 0$ and $N = k - 1$) result in nearly the same speedup and cost, while setting $N = 1$ improves both the speedup and cost. As shown, $N = k/2$ results in the optimal speedup and cost, which are comparable to the speedup and cost when using the optimal number of backup replica tasks (see Figure 4.6). Since $N = 0$ represents using BSP on transient servers, partial barriers offers a clear advantage in terms of speedup and cost, similar to using backup replica tasks.

Compared to using backup replica tasks, partial barriers offer a slightly lower maximum speedup ($\sim$2.5$\times$ versus $\sim$3.5$\times$) for a marginally lower cost ($\sim$0.75$\times$ versus $\sim$1$\times$). However, one advantage of partial barriers over using backup replica tasks is that the latter is speculative, and requires jobs to first identify slow tasks, while the former is not. One disadvantage of partial barriers is that it may require algorithmic and implementation changes, since it alters the synchronization model.

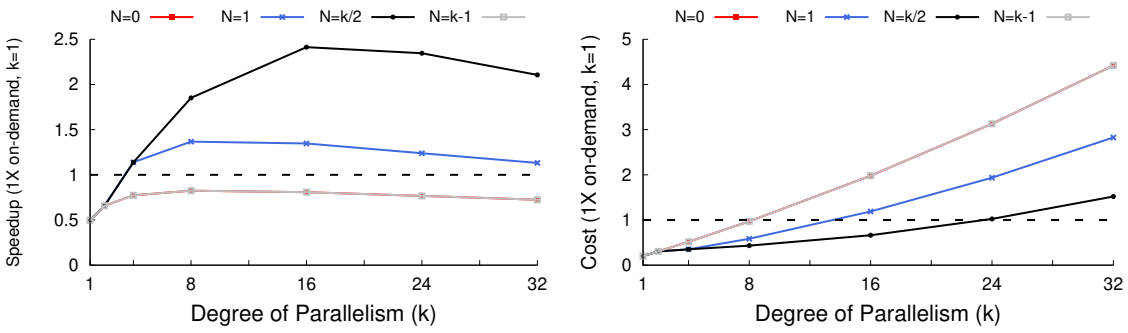


Figure 4.10: The speedup (top) and cost (bottom) of executing our representative parallel job using partial barriers for different numbers of dropped slow tasks N as the degree of parallelism increases.

Result: *Increasing the degree of parallelism k when using transient servers with partial barriers strictly improves the speedup and cost compared to BSP, and enables speedups greater than* 1. *The approach has a comparable speedup and cost as using backup replica tasks.*

4.3.6 Flexible Synchronization on Transient Servers

Recent work has introduced a flexible synchronous processing model [85]. FSP proposes a synchronization model that initiates synchronization barriers dynamically based on the progress of the tasks. Thus, if it identifies stragglers, FSP can dynamically initiate a synchronization barrier, and allow the fast tasks to continue execution. We model FSP similar to partial barriers, but where jobs do not have to re-execute the work of "dropped" tasks at each barrier. In this case, once $k - N$ tasks have reached a barrier, the job initiates synchronization among all k tasks, allowing all tasks to proceed past the barrier. The remaining work of these slow tasks is then re-distributed across all the servers. We note that the original FSP model *does not* redistribute work, since it targets "naturally occurring" stragglers that are temporary and rare. Since stragglers due to transient servers are expected and frequent, we must distribute each interval to gain any speedup. Thus, using FSP in practice on transient servers would require some changes. As with partial barriers, prior work on

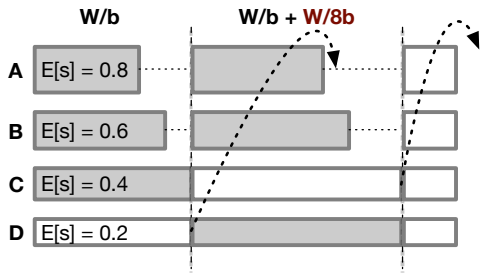


Figure 4.11: Parallel job using FSP on transient servers.

FSP uses a more sophisticated approach that dynamically determines barrier points by monitoring task progress.

Figure 4.11 illustrates using FSP on transient servers with $N = 1$. The figure is the same as with partial barriers (Figure 4.9) except that the additional work shifted to the next barrier interval is lower, since not all the work has to be redone. In both cases, the expected case is that half the work assigned to the N slowest servers is completed. Thus, the expected running time below is equivalent to that of partial barriers (from Equation 4.10), except that FSP only has to shift and re-distribute the remaining half of the work to be completed in the next barrier interval. This is the reason for the additional 2 in the denominator compared to Equation 4.10.

$$T = \frac{W(1+k)}{b \times k \times (1+N)} \times [\sum_{i=2}^{b}[b-(i-1)](\frac{N}{2k})^{i-2}] + \frac{W(1+k)}{b \times k} \times [\sum_{i=1}^{b}(\frac{N}{2k})^{i-1}] + (n \times b \times k) \quad (4.11)$$

Figure 4.12 shows the speedup (left) and cost (right) of using FSP on transient servers for different values of N. Of course, FSP is strictly better than partial barriers because it is equivalent, except that it does not waste resources re-executing work. In addition, FSP offers a maximum speedup near that of using backup replica tasks, but for a lower cost (see Figure 4.5).

While FSP offers the best performance and cost, it does pose some challenges. In particular, FSP was developed in the context of distributed ML, and, as with

bounded staleness and partial barriers, may impact algorithmic convergence time and accuracy, which we do not model. In addition, as with partial barriers, FSP is a new synchronization model that likely requires algorithmic and implementation changes. Prior work has only applied FSP to specific problems, e.g., Expectation-Maximization (EM) [85] and Stochastic Gradient Descent [93]. As a result, FSP's generality is not yet clear.

Result: *Increasing the degree of parallelism k when using transient servers with FSP yields the best speedup and cost among the straggler mitigation techniques we model.*

4.3.7 Summary

Our analysis shows that users must jointly consider both speedup and cost when deciding whether and how to use transient servers for parallel jobs. While different users may value speedup and cost differently, Figure 4.13 plots the speedup/cost ratio for all of the straggler mitigation techniques above as the degree of parallelism k increases Since a large speedup and a low cost are preferable, higher values of the speedup/cost ratio indicate a better "bang for your buck" for parallel jobs. Interestingly, using BSP with on-demand servers is not the worst option, as using BSP with transient servers and no replicas has a lower speedup/cost for all but the smallest values of k, despite their high discount. Using partial barriers and using backup replica tasks offer a speedup/cost ratio in the middle. In this case, we use parameter values of $N = k/2$ and $r = k$, respectively, which yield the maximum speedup/cost ratio for these techniques. At lower values of k, backup replica tasks offer a higher speedup/cost ratio, while at larger values of $k \geq 16$ partial barriers yield a higher ratio.

As mentioned above, using FSP with transient servers yields the highest speed/cost ratio across all values of k. In addition, since the use of backup replica tasks is not mutually exclusive to using FSP, we were interested in whether com-

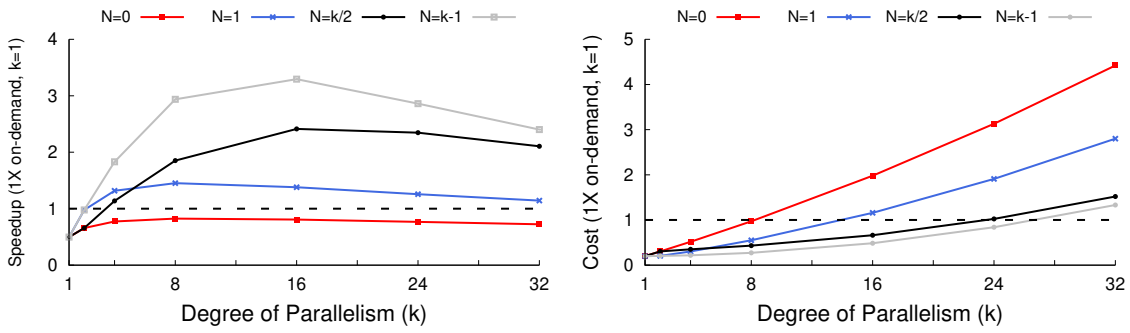


Figure 4.12: The speedup (left) and cost (right) of executing our representative parallel job using flexible synchronous processing (FSP) for different numbers of dropped slow tasks N as the degree of parallelism increases.

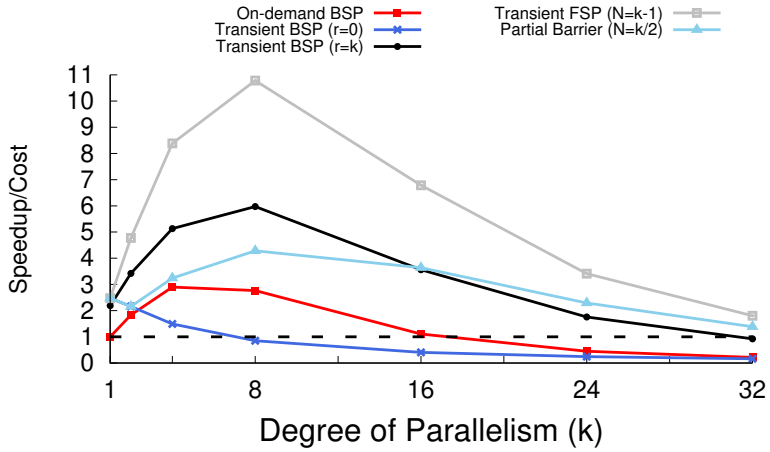


Figure 4.13: The speedup/cost ratio for executing our baseline parallel job for different straggler mitigation techniques on on-demand and transient servers.

bining these techniques offered any advantage. We omit the equation for T for this hybrid technique, as it is complex, but show the result in Figure 4.14 for different combinations of N and r. In all cases, we set $N = k - 1$, since it is optimal, as there is no reason for any task to ever wait with FSP. The result shows that using backup replicas in combination with FSP *does not* offer an advantage in overall speedup/cost. Replicas only add an additional cost, but offer no advantage in terms of running time, since FSP need not wait on stragglers anyway.

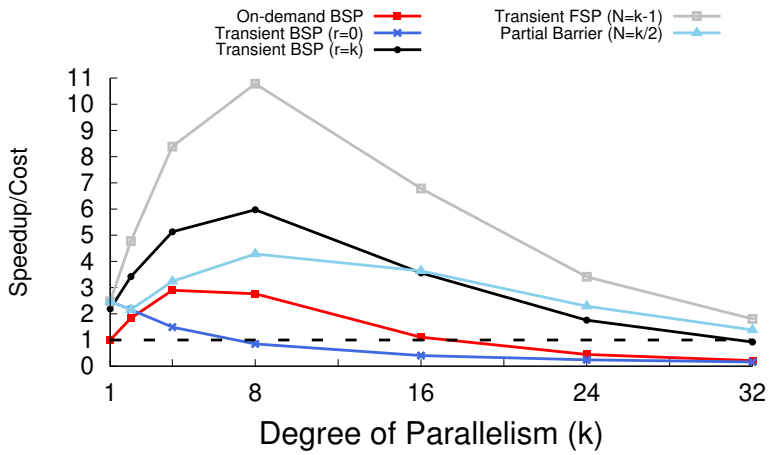


Figure 4.14: The speedup/cost ratio for executing our baseline parallel job for a hybrid straggler mitigation technique that combines FSP with backup tasks.

4.4 Conclusion and Status

In this chapter, we analyze the speedup and cost of executing parallel batch jobs, such as distributed ML jobs, on highly discounted transient cloud resources using many different straggler mitigation techniques. We do so in the context of a simple probabilistic model for transient server performance. Using this model, we derive the expected running time and cost for straggler mitigation techniques proposed in prior work for a simple parallel job with synchronization barriers. A key difference between our work and prior work on straggler mitigation is our focus on cost, rather than performance, on cloud platforms. Our analysis shows that i) transient servers offer complex tradeoffs compared to using on-demand servers, and can result in higher overall costs despite their highly discounted price due to their probabilistic performance; ii) common approaches to straggler mitigation, which is a well-studied problem, are less effective using transient servers that cause frequent and severe stragglers; and iii) a recent approach to flexible synchronization [85, 93] offers the best speedup per cost across all the techniques we study.

Status. Additional details on the analysis, and models are in [22].

CHAPTER 5

OPTIMIZING LONG-TERM BATCH WORKLOADS ON MIXED VM PURCHASING OPTIONS

Cloud platforms offer the same VMs under many purchasing options that specify different costs and time commitments, such as on-demand, reserved, sustained-use, scheduled reserve, transient, and spot block. In general, the stronger the commitment, i.e., longer and less flexible, the lower the price. However, longer and less flexible time commitments can increase cloud costs for users if future workloads cannot utilize the VMs they committed to buying. Large cloud customers often find it challenging to choose the right mix of purchasing options to reduce their long-term costs, while retaining the ability to adjust capacity up and down in response to workload variations. In this chapter, we design policies to optimize long-term cloud costs by selecting a mix of VM purchasing options based on short- and long-term expectations of workload utilization.

5.1 Policies Overview

Given a set of cloud VM types (of different fixed sizes) offered under the purchasing options in Chapter 2, our problem is to select the resources and purchasing options that minimize the long-term cost based on both short- and long-term expectations of workload utilization. We assume our workload is composed of batch job submissions from users that include the requested number of cores and memory.

To simplify the problem, we first consider an optimistic optimal offline approach which assumes perfect knowledge of the future workload as well as the ability to allocate fractional demand to fractional resources where possible. That is, even though

our workload's demand is composed of discrete jobs, we assume that discrete jobs can be sub-divided across resources, and purchasing options. Similarly, even though cloud VMs are mostly composed of discrete resource bundles (types), we assume resources (cores and memory) can be purchased separately and bundled together in any quantity. Solving this offline case provides an *optimistic* optimal upper bound on the realizable cost savings in practice. We then present our online approach, which removes the assumptions above of perfect future knowledge and fractional demand. Our online policy is similar to the offline policy, but substitutes imperfect predictions of short- and long-term demand (based on historical data) for perfect knowledge and considers the availability of limited VM types.

5.2 Optimistic Optimal Offline Approach

We model the workload trace in terms of the aggregate resource demand per unit time from all active jobs within that time unit; the aggregate resource demand is defined to be the total cores and memory requested by all active jobs within a time unit. Thus, the workload can be viewed as a time-varying function of resource demand. The intuition for our optimal offline approach is as follows: for each unit of resource demand, e.g., cores and memory requested by jobs, we compute the cost of the necessary resources under each purchasing option to satisfy that unit of demand normalized by its utilization over the length of the commitment. For example, for one unit of resource demand and a 1-year reservation, we normalize the cost based on the utilization over a year. Thus, if the reservation's cost is 60% of the on-demand cost, but the utilization over the year is only 60%, then its normalized cost is the same as the normalized on-demand cost. Given the normalized costs for various options, we select the cheapest option for each unit of resource demand until the demand across time slots is satisfied.

Our general strategy applies directly when considering the relative cost of on-demand, 1- and 3-year reserved, scheduled reserved, and the sustained-use options. In these cases, we can directly compute a optimistic optimal normalized cost for the resources to satisfy each unit of demand under our assumption of a fractional supply and demand. However, the normalized cost of the transient and spot block options is *directly* a function of each job's length, which prevents us from directly computing it under the assumption of a fractional resource demand. For example, for the transient option, the longer the job, the more revocations it will experience and the greater its normalized cost. Similarly, a job that runs for 3 hours on a 3-hour spot block resources has a higher normalized cost than a job that runs for 1 hour on a 1-hour spot block resource. Thus, when computing the normalized cost for these options, we *must* consider job length. As a result, in the offline case, we first assign a normalized cost for using the transient and spot block for each job, as discussed below, before considering the other purchasing options.

Transient. As prior work has discussed [66,74], the normalized cost of using transient VMs is a function of not only their relative cost per unit time and the job's length, but also the revocation rate and the use of fault-tolerance mechanisms to mitigate the impact of revocations. In general, the longer a job, the more likely it is to experience a revocation. However, precise revocation rates (and their distribution) are not publicly known, and likely differ across providers.

Our analysis assumes a more basic use of transient VMs that assumes no checkpointing by restarting a job after each revocation. In particular, to ensure a job assigned to a transient VM completes, once it has experienced a revocation, we just restart it on an on-demand VM. Under this simple model, we compute the expected cost $E[C(T)]$ to execute a job of length T using the transient option as below. We assume that $p_{\texttt{transient}}$ and $p_{\texttt{ondemand}}$ are the relative transient and on-demand prices,

$R(T)$ is the probability the job will be revoked before it completes at time T, and $E_{\texttt{revoke}}[T] < T$ is the expected time a revoked job runs.

$$\begin{aligned} E[C(T)] = (1 - R(T))(p_{\texttt{transient}} \times T) + \\ R(T)(p_{\texttt{transient}} \times E_{\texttt{revoke}}[T] + p_{\texttt{ondemand}} \times T) \end{aligned} \tag{5.1}$$

The first term represents the total cost to run the job if it is not revoked, while the second term represents the total cost to run the job if it is revoked. The normalized cost per unit time is then the expected cost to execute the job $E[C(T)]$ divided by the expected running time, which is $(1 - R(T)) \times T + R(T) \times (E_{\texttt{revoke}}[T] + T)$. Of course, our approach is dependent on the revocation characteristics. For the offline case, we assume we know each job's running time, as well as the revocation characteristics, and can directly compute the normalized cost per job per unit time.

Spot Block. Spot blocks can be purchased in 1-, 2-, 3-, 4-, 5-, or 6-hour increments with a higher discount applied to shorter increments. In the offline case, since we know the duration of each job, we simply map jobs to the smallest spot block increment that is greater than their running time, and compute the corresponding normalized cost per unit of time based on the increment's discount. Jobs longer than 6 hours cannot be run using the spot block option. As we discuss, for the online case, we must predict each job's running time based on historical data to compute this normalized cost.

On-demand. Computing the normalized cost for on-demand VMs is straightforward: we simply assign the on-demand cost to each unit of resource demand.

Sustained-Use. The sustained-use discount for an on-demand server applies regardless of when a unit of resource demand is used within a month. Thus, to compute it, we need only compute the average resource demand over each month-long billing period. The full discount applies to the floor of this average, while a partial discount

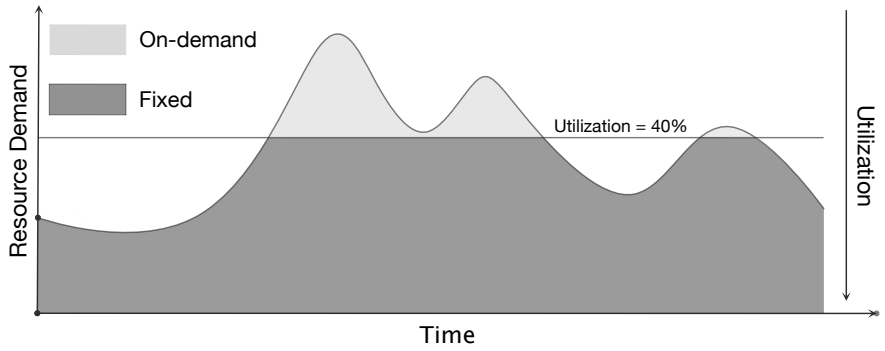


Figure 5.1: Illustration of the utilization of each unit of resource demand for normalizing the reserved option cost.

applies to the remainder of the average. Note that the sustained-use discount applies to the on-demand cost, so, when applied, it always results in a normalized cost equal to or less than the on-demand cost.

Reserved. In the offline case, when normalizing the cost of the reserved option, we assume a fractional resource, as mentioned above, that aggregates the resource demand of all jobs. Figure 5.1 shows an illustrative example of the aggregate resource demand over time for a batch trace. For each unit of stacked resource demand starting at 0 on the y-axis, we compute the utilization of that unit of demand over both the 1- and 3-year offered reserved terms. For the 1- and 3-year terms we assign the same normalized cost per unit of time to each unit of stacked resource demand starting at 0 based on its utilization across the term. This has the effect, as in Figure 6.1, of drawing a line at each unit of demand on the y-axis, computing its utilization, and then normalizing its cost across the term length based on the utilization. Since the 1- and 3-year terms are 60% and 40% of the on-demand cost as shown in Table 2.1, the resource utilization must be at least 60% and 40%, respectively, to have a normalized cost less than the on-demand cost. The higher the utilization, the lower the normalized cost, such that 100% utilization yields the entire discount.

Scheduled Reserved. Computing the normalized cost of the scheduled reserve option is similar to that of reserved, in that we normalize the cost relative to the utilization over the scheduled reserved's 1-year term. However, unlike with the re-

served option, which is essentially a single schedule, with scheduled reserved we must consider every possible daily, weekly, and monthly schedule. There may be multiple non-overlapping schedules for each unit of demand over a 1-year term that yield a lower cost than either the on-demand or reserved price.

To find these schedules, we observe that the problem reduces almost directly to the classic weighted job scheduling problem, which has an efficient and well-known dynamic programming solution that runs in $O(n \log n)$ time (and is often used in tutorials to teach dynamic programming[1]). The weighted job scheduling problem takes as in put a set of n jobs that each have a start and end time, as well as an associated value. Given multiple, possibly overlapping jobs in time, the problem is to select the non-overlapping set of jobs, such that we maximize overall value. In our approach, each possible multi-hour scheduling interval is akin to a job, the normalized discount of that schedule is akin to the value, and the output is the cheapest set of schedules. Thus, we can solve the problem directly, given the normalized cost for each schedule.

Selecting Purchasing Options. To compute our optimistic offline optimal cost, we select the lowest cost for each unit of stacked resource demand, starting at 0. We first determine the lowest of the normalized transient, spot block, on-demand, and scheduled reserved options for each unit of resource demand for each unit of time before considering the reserved options. For the reserved options, we first consider the 1-year option. We compute the average cost of the lowest cost non-reserved options above for each unit of resource demand over each 1-year term. Since we can purchase a reserved option at any time, we use a 1-year sliding window that performs this comparison over each 1-year interval in our data. Assuming there is >3 years of data available, we next apply the same approach to compute the normalized 3-year

[1] `https://www.geeksforgeeks.org/weighted-job-scheduling-log-n-time/`

reserved cost. We compare this normalized 3-year cost with that of the lowest of the 1-year reserved option and non-reserved options, and take the lowest value. We can apply a sliding window depending on data availability. Our approach above will yield the lowest cost purchasing option for each unit of stacked resource demand over time under our assumptions of perfect future knowledge and fractional supply and demand.

5.3 Practical Online Approach

Since our optimistic assumptions for the offline approach are not practical, we adapt it to an online approach and evaluate it using 4 years of job submission data from a large-scale batch cluster. Our practical online approach is essentially the same as our offline approach, but utilizes predictions of short- and long-term demand in the place of perfect knowledge, and does not assume a fractional supply and demand. Since our predictions are imperfect, our online approach is a heuristic. However, even given perfect future knowledge of the workload, the problem is NP-hard, as removing the fractional assumption makes it strictly harder than the NP-hard bin packing problem.

Our predictions of long-term demand are straightforward: we simply take prior job submission data and apply our offline approach from the previous section to estimate the amount of 1-year, 3-year, and scheduled reserved capacity to purchase. While evaluating we make these decisions based on the first year of job data, and evaluate our online approach over the next 3 years. Since we do not have 3 years of prior data, we simply assume our training year will repeat to estimate the 3-year reserved capacity to purchase. The accuracy of such long-term predictions is a function of whether a workload changes significantly over time. Our goal here is not to develop the most accurate prediction model, rather it is to quantify the long-term cost benefits of mixing different contract types using reasonable predictions.

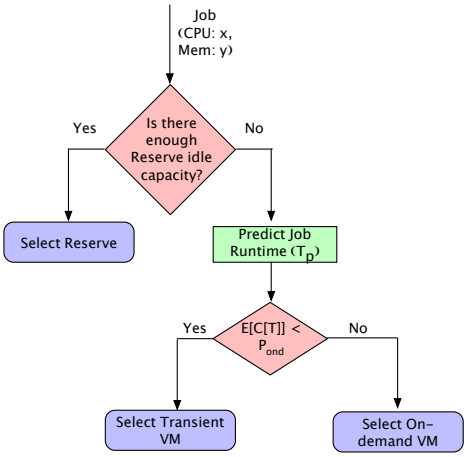


Figure 5.2: Simple flow chart for selecting the VM purchasing option online when only reserved, transient, and on-demand are available, as with Microsoft.

The offline approach separately determines the amount of cores and memory to purchase under each of the reserved purchasing options. In practice, we must map these cores and memory to specific types of VMs. To do this, we simply purchase the largest VM types available that have a ratio of cores to memory that is closest to the offline ratio for each purchasing option.

After purchasing reserved capacity at the outset, as jobs arrive online, we schedule them on available reserved capacity based on their requested cores and memory. If there are no available resources to execute a job, we dynamically acquire additional non-reserved resources to execute the job. Since we are using the cloud, our batch system is not limited to a fixed-size cluster, and thus jobs never need wait in a queue for resources. When dynamically acquiring non-reserved resources, we must determine whether to purchase on-demand, transient, or spot block.[2]

Given a prediction of the job's running time (as well as the transient revocation characteristics), we compute the normalized cost of each option and select the lowest cost. We compute this normalized cost for every available VM type based on the job's requested cores and memory. The lowest cost VM type is generally the smallest one

[2]The sustained-use discount is always automatically applied to on-demand when used, so we need not consider it here.

that has the requisite cores and memory, although a discrete set of VM types results in wasted resources that increase the normalized cost. Figure 5.2 depicts a simple flow chart for the online case where only reserved, transient, and on-demand options are available (as with Microsoft). Here, $E[C[T]]$ is the normalized cost to execute a job of length T on a transient VM, while P_{ond} is the expected cost to execute it on an on-demand VM.

Job Runtime Predictions. We develop a simple regression model based on a year of historical job submission data to predict job runtime. As above, our goal is not to develop the most accurate job runtime prediction model, but to quantify the long-term cost benefits of mixing different contract types using reasonable predictions. Each job in our batch trace, lists a user ID, job submission time, requested cores and memory, and maximum runtime limit. The maximum runtime limit is supplied by the user and represents the maximum time the job can run before the system kills it—it is not a job runtime estimate. We use these attributes as the input features to a regression model with the job runtime as the output variable. Once trained, the model supplies a job runtime prediction given a new job's input features.

5.4 Implementation

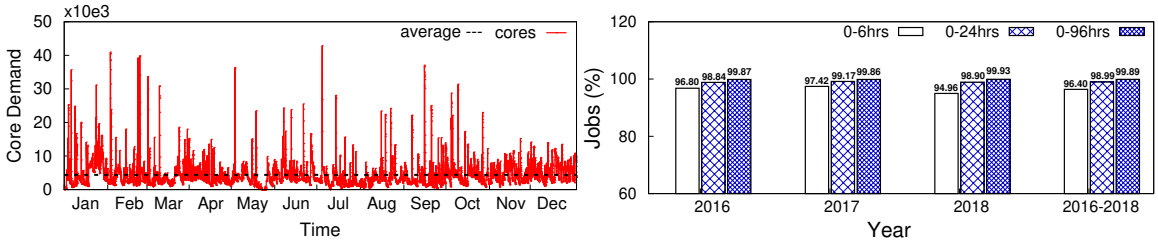
We implemented both the optimistic offline approach (§5.2) and practical online approach (§5.3) in Python. The offline implementation takes as input a trace of job submissions, and uses it to compute the mix of VM purchasing options that minimize the cost based on the assumptions in §5.2. Each job entry includes its submission time, requested number of cores and memory, and running time. The online implementation also takes as input a prior year's trace of job submissions, and uses it to determine the amount of 1-year, 3-year, and scheduled reserved capacity, assuming subsequent years will be similar to the prior year. The online implementation also regresses on this data to build its job runtime prediction model.

We evaluate our approach in simulation using a 4-year trace of job submissions from a 14k batch cluster for a major state University system (serving multiple campuses). In addition to the job submission time and requested cores and memory, each job entry also includes a user ID and maximum running time limit. For job runtime predictions, we use ridge regression using these 4 input features and a job's actual running time as the output feature. For the online approach, we use the first year of jobs (2015) for training, and then evaluate on the next 3 years (2016-2018) of job submissions. We evaluate the offline approach on the same 3 years.

5.5 Evaluation

Our evaluation examines the cost benefits of using a mix of VM purchasing options in both the offline and online cases compared to using a single purchasing option, either all on-demand or all reserved, for our batch trace. We examine the cost benefits for the set of purchasing options offered by each cloud provider. Specifically, Microsoft offers on-demand, 1- and 3-year reserved, and transient. Google offers the same as Microsoft but also with a sustained-use discount and a customized option, while Amazon also offers the same as Microsoft but with scheduled reserved and spot block. In addition, Google's variant of the transient option has a maximum lifetime of 24 hours, while Amazon and Microsoft's variant has no maximum lifetime.

Since our focus is on the benefits of different purchasing options, we use the same standard set of VM types and prices across all providers. As a result, our evaluation does not reflect the absolute cost difference between providers, but the relative benefits of each provider's set of purchasing options. We consider standard VM types with 1, 2, 4, 8, 16, 32, and 64 cores with 4, 8, 16, 32, 64, 128, and 256 GB memory, respectively. We assume the cost of a 1 core, 4GB VM is $0.0481 per hour, which is equivalent to an `m5.large` VM in Amazon with larger capacity VMs priced as a simple scalar multiple.



(a) Resource Demand (b) Job Runtimes

Figure 5.3: (a) The hourly core demand over 2018. The average over the year is 4380 cores. (b) Job runtime for different length jobs each year in our batch trace.

5.5.1 Batch Trace Characteristics

Figure 5.3 (a) shows the hourly core demand on average for our batch cluster over the year 2018. While our cluster has 14k cores, the core demand peaks at nearly 43k cores, indicating that jobs may periodically experience long waiting times. In the cloud, these waiting times are not necessary as there is no resource constraint. As might be expected, the average core utilization over the year is much less than the peak, at only 4380 cores, resulting in a 31% average utilization. Figure 5.3 (b) shows that a high percentage (>96%) of the jobs are less than 6 hours in length, and only a small percentage are longer. The number of jobs that run longer than 96 hours is quite small at 0.11%.

5.5.2 Mixing VM Purchasing Options

Below, we evaluate the cost savings from mixing all the VM purchasing options in both the online and offline case.

Optimistic Offline Approach. Figure 5.4 shows the cost of mixing VM purchasing options using our optimistic offline approach from 2016-2018 for the sets of VM purchasing options offered by the different cloud providers. Figure 5.5 shows the average percentage mix of each purchasing option over the 3-year period. We see that the cost relative to on-demand is 35% for Amazon and Microsoft, but only 41% for Google-Standard, which includes the sustained-use discount but does not permit the

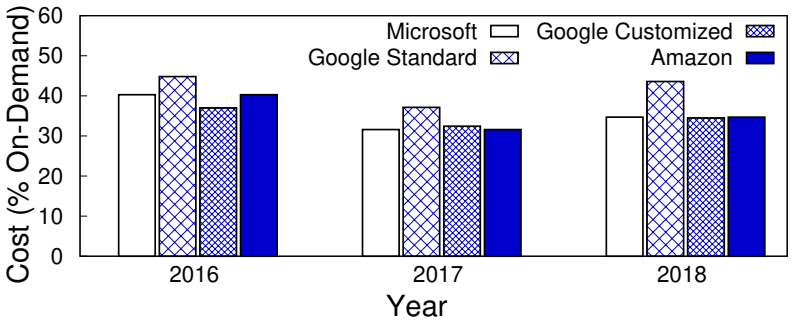


Figure 5.4: Cost for executing our batch trace using all purchasing options from the different cloud providers in the optimistic offline case as a percentage of using on-demand only.

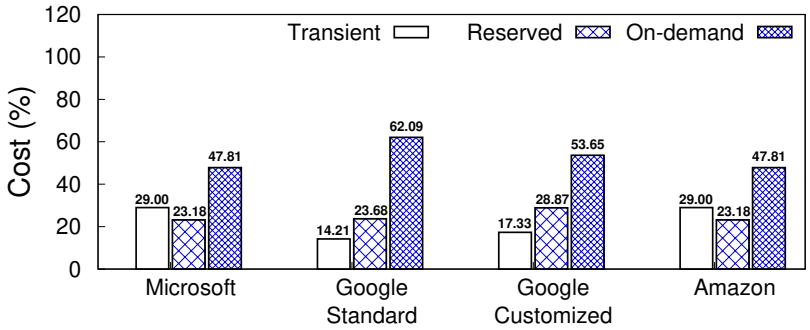
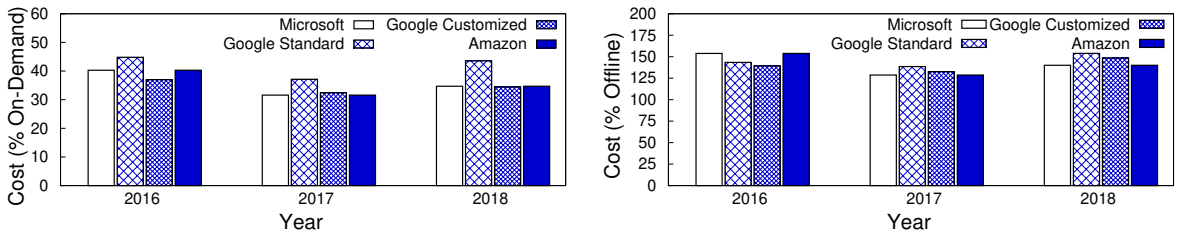


Figure 5.5: Mix of VM purchasing options used over 2016-2018 in the offline case (with the transient option).

customized option. The savings for Google-Customized then drops to 33.62% of the on-demand cost.

The Amazon and Microsoft cases are the same because Amazon's additional options—spot block and reserved—are never used in the offline case, and we treat the transient case the same for both Microsoft and Amazon. Spot blocks never offer a cost benefit for short 1-6 hour jobs over transient, although they may be used in practice based on a job's requirements, i.e., if it cannot risk a revocation. Scheduled reserved is never selected, largely because its discount is too low (5-10%). Google's set of purchasing options is cheaper due to both their sustained-use discount and customized option. Adding the sustained-use discount in Google-Standard results in a cost 41% the on-demand cost, while also adding the customized option reduces



(a) (b)

Figure 5.6: Cost for executing our batch trace using all purchasing options from the different cloud providers in the online case as a percentage of using on-demand only (a) and optimistic offline cost (b).

this to 33.62%, despite the 5% increase in price. Nearly every job makes use of the customized option, as most are not within 5% of the size of a standard VM.

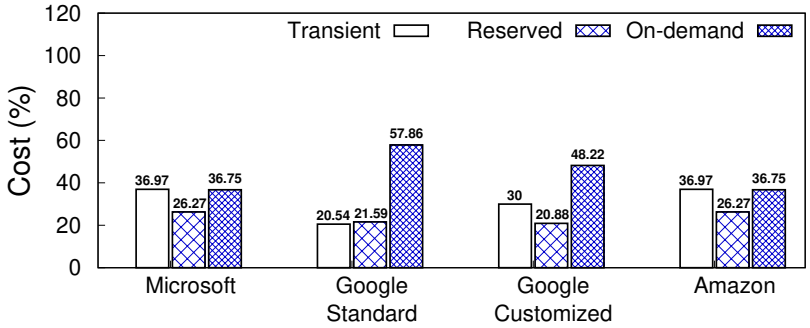
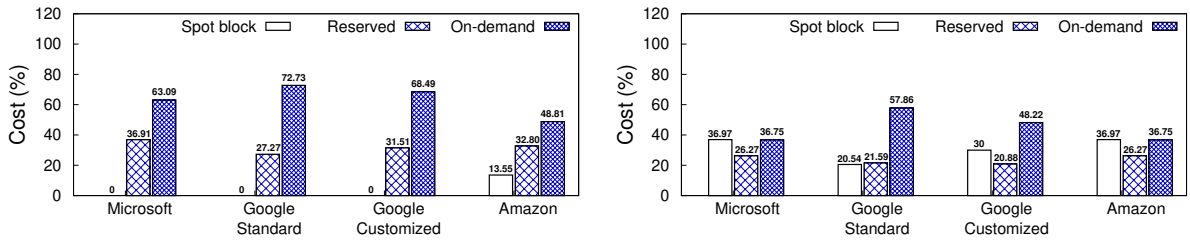


Figure 5.7: Mix of VM purchasing options used over 2016-2018 in the offline case (with the transient option).

Practical Online Approach. Figure 5.6 shows the cost of our online approach both as a percentage of the cost of only using the on-demand option (a), and as a percentage of the optimistic offline cost (b). Figure 5.7 shows the average percentage of each VM purchasing option over the 3-year period. The mix of purchasing options is similar to the offline approach above, although the percentage of transient VMs used decreases due to inaccurate job runtime predictions, as we discuss below. As above, the online approach never selects the spot block option or uses scheduled reserved. As a result, both Amazon's and Microsoft's results are the same in the online case as well. Also as before, Google-Standard and Google-Customized results in slightly



(a) Offline Case (b) Online Case

Figure 5.8: Mix of VM purchasing options used over 2016-2018 in the (a) offline case and (b) online case without the transient option.

lower relative costs in (a). Figure 5.6(b) compares the online approach for each set of purchasing options with the respective offline approach. Google's online cost is the highest relative to its offline cost due to incorrect job runtime predictions combined with its short maximum lifetime for transient VMs. The online approach results in 35% greater cost compared to their respective optimistic offline approach for Amazon and Microsoft, while it results in 55% greater cost for Google.

5.5.3 Removing Transient VMs

We repeat the evaluation above, but in this case without the transient option. We do this for two reasons. First, transient characteristics are not publicly known, and have a significant impact on our results. Second, not all jobs can run on transient VMs due to having strict deadlines or not being able to gracefully handle revocations.

Optimistic Offline Approach. Figure 5.8 (a) shows the cost of mixing VM purchasing options without the transient option in the optimistic offline case. The figure shows that the overall cost increased relative to with the transient option because transient VMs were by far the cheapest option. However, this does not occur to a significant degree, primarily because not a large fraction of the CPU-hours come from jobs that are less than 6 hours in length (see Figure **??**(b)), which mitigates the impact of spot block. However, the availability of spot block for Amazon with no low cost replacement for transient at either Google or Microsoft results in Amazon

having the lowest overall cost. Their cost savings exceed Google even when using the sustained-use discount and the customized option.

Practical Online Approach. Figure 5.8 (b) then shows the same results when using our online approach as a percentage of the on-demand cost. The benefits of spot block decrease when using the online approach, since the predictions of job running times are not accurate. Due to the reduced benefit of spot block, Google's sustained-use discount and customized option enable it to achieve the lowest cost.

5.6 Related Works

There has been significant prior work in optimizing particular workloads and applications for using different VM purchasing options. However, most of this work focuses on optimizing a single type of purchasing option and comparing it with using on-demand, rather than looking at all of them in combination. Perhaps most relevant to our work is HCloud [34], which focuses on combining the reserved and on-demand purchasing options. However, HCloud primarily focuses on determining how to map jobs to on-demand and reserved VMs based on their sensitivity to performance interference. As a result, HCloud focuses on small-scale workloads and does not evaluate their approach using a real large-scale workload over a multi-year period, nor optimize for other purchasing options.

There is a large body of work on optimizing for other purchasing options, including transient VMs [63, 64, 74, 89, 90], the sustained-use discount [94], and burstable VMs (which we consider to be a different VM type) [82]. There is also a large body of related work on selecting the appropriate VM type for a particular application or workload [16, 47, 80, 88]. Our work differs from this work in that we focus on selecting the best purchasing option given accurate knowledge of a job's resource requirements.

5.7 Conclusion and Status

Cloud platforms offer the same VMs under a variety of purchasing options that specify different costs and time commitments, such as on-demand, reserved, sustained-use, scheduled reserve, spot/preemptible, and spot block. Choosing from among these options can be challenging. To address this problem, in this work, we design policies to optimize long-term cloud costs by selecting a mix of VM purchasing options based on short- and long-term expectations of workload utilization. We evaluate our policies on a batch job trace spanning 4 years from a large shared cluster for a major state University system that includes 14k cores and 60 million job submissions, and show how these jobs could be cost-effectively executed in the cloud using our approach. Our results show that our policies incur a cost within 41% of an optimistic offline optimal approach, are 50% less than solely using on-demand VMs, and 79% less than using reserved VMs.

Status Additional details on the analysis, and models are in [17].

CHAPTER 6

OPTIMALLY PROVISIONING FIXED RESOURCES FOR CLOUD-ENABLED SCHEDULERS

While cloud platforms enable users to rent computing resources on demand to execute their jobs, buying fixed resources (or reserved VMs) is still much cheaper than renting if their utilization is high. Thus, optimizing cloud costs requires users to determine how many fixed resources to buy versus rent based on their workload. In this chapter, we introduce the concept of waiting policy for the cloud-enabled schedulers and show that optimizing the cloud costs depend on it. We define multiple waiting policies and develop simple analytical models to reveal their key tradeoffs between fixed resource provisioning, cost, and job waiting time.

6.1 Motivation

Cloud platforms enable users to rent computing resources on demand, in the form of virtual machines (VMs), to execute their jobs. Cloud-enabled infrastructure uses similar software systems as private clusters to manage resources at large scales such as Slurm [8] or Kubernetes [30]. Users submit jobs, with specified resource requirements, to these schedulers, which either allocate idle resources to execute them or force them to wait for idle resources to become available. Typically private clusters are sized for their peak demands, and as a result they often have low average utilization (<30%) but may periodically experience large bursts in job arrivals that result in long job waiting times.

As job schedulers migrate to the cloud, they have many options for optimizing cost and reducing job waiting times. Acquiring on-demand VMs dynamically allows users

to pay for the resources only when jobs need them and hence cloud's operating costs are often much lower than the capital cost of an under-utilized fixed-size cluster, since the latter must effectively "pay" when resources are idle. In addition, cloud provides the illusion of infinite scalability, where jobs may never need to wait for the resources.

Importantly, however, buying fixed resources (or reserving them for long periods) is significantly cheaper than renting resources on demand if the fixed resources are highly utilized. Cloud pricing models in 2 make this clear, as reserving a VM for 1-3 year costs 40-60% less per-hour than renting an equivalent on-demand VM over the same time period. For example, reserving a `m5.large` VM from Amazon Web Services (AWS), which includes 2 cores and 8GB RAM, for 3 years currently costs $988, while renting it on demand costs $0.096/hour or $2,522.88 over the same period. Of course, fixed resources are only cost-effective if they are highly utilized: if jobs only execute on the `m5.large` for less than a third of the time, the on-demand option is cheaper (at a cost of $840.96). Thus, a mixed infrastructure that satisfies some baseload with highly-utilized fixed resources, and satisfies load bursts using on-demand resources can decrease cost. Notably, hybrid clouds, which combine fixed private resources with cloud bursting, use this approach [41, 57], as do many companies, which both buy reserved VMs and dynamically rent on-demand VMs [45].

6.2 Introduce Waiting Policy

A waiting policy is the dual of a scheduling policy: while a scheduling policy determines which jobs run when fixed resources are available, a waiting policy determines which jobs wait for fixed resources when they are not available (rather than run immediately on on-demand resources). Waiting polcy for cloud-enabled scheduler determines the number of fixed resources to provision in order to optimize the cost. While there has been decades of work on job scheduling policies, we know of no prior work that defines or analyzes waiting policies, which are distinct from scheduling poli-

cies in that cloud-enabled schedulers define both independently of each other. For cloud-enabled schedulers, the waiting policy is important, since it dictates the tradeoff between job performance and cost. Our hypothesis is that, by optimizing their waiting policy, cloud-enabled schedulers can significantly reduce job waiting times, while mitigating the impact on cost, or vice versa.

Intuitively, the longer jobs are willing to wait for fixed resources, the higher their utilization, and the lower their overall cost. However, as we show, the relationship and tradeoff between the number of fixed resources, the waiting policy, and the optimal cost is non-intuitive. To better understand these tradeoffs, we define multiple fundamental non-selective and selective waiting policies and develop simple analytical models for them. Non-selective waiting policies apply the same policy to all jobs, while selective waiting policies apply the policy to only selected jobs based on system or job characteristics. Our modeling approach extends classic marginal analysis by combining it with a number of different queuing results and analyses to model cloud cost under job waiting.

6.3 Background: Marginal Analysis

In economics, marginal analysis examines the additional benefits of some activity compared to the additional costs incurred by that activity. Determining the optimal mix of fixed and on-demand resources to execute a workload on a cloud platform to minimize cost is a classic marginal analysis problem. Given a workload and some fixed resources capable of servicing a fraction of it, the marginal analysis problem is to determine whether the additional benefit of acquiring one more fixed resource to serve (a portion of) the remaining workload outweighs its cost, i.e., the savings from renting an on-demand VMs to service the same portion.

Figure 6.1 illustrates marginal analysis pictorially for an example workload where time is on the x-axis and resource demand is on the y-axis. We assume the fixed

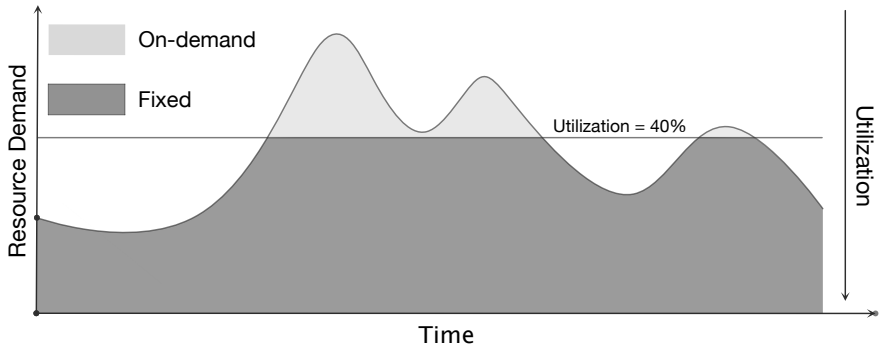


Figure 6.1: Illustration of utilization for each unit of stacked resource demand and the break even point at 40% utilization.

(reserved 3yr) and on-demand VMs have the same prices as in Table 2.1. To determine the optimal mix of fixed and on-demand resources using marginal analysis, we simply add fixed resources, one at a time, to satisfy each unit of stacked resource demand in order (starting from 0 on the y-axis) up to the point where the utilization of the reserved VM equals our break even point on the y-axis, which is 40% (in dark grey). When the instantaneous demand exceeds the total reserved VMs capacity at the horizontal line (in light grey), dynamically acquiring and releasing on-demand VMs to satisfy the remaining workload is cheaper.

More formally, let p_f and p_o denote the price per unit time for a fixed resource (at 100% utilization) and on-demand VM, respectively, let d denote the discount factor for a reserved VM, such that $p_f{=}d \times p_o$ and $0{\leq}d{\leq}1$, and let T denote the workload's duration. The cost of adding one more fixed resource s over the workload's duration T is $p_f \times T$. Now suppose this s^{th} resource operates at utilization ρ_s when servicing the remaining workload. Since the scheduler can acquire and release on-demand VMs at any time, the cost of servicing the remaining workload using an on-demand VM is $\rho_s{\times}T{\times}p_o$, as the scheduler can acquire the on-demand resource in $\rho_s{\times}T$ time slots and release it when idle. Thus, using a reserved VM is only cheaper if $p_f \times T < \rho_s \times T \times p_o$. By substituting $p_f{=}d \times p_o$, we observe that only when $d{<}\rho_s$, or the discount factor is less than the utilization of the last reserved VM we added, is acquiring an additional reserved VM cheaper than using on-demand VMs. Similarly, the cost of provisioning

an additional reserved or on-demand VM is equal when ρ_s=d, or the discount factor equals the utilization of the last fixed resource. Beyond this break even point, there is no marginal cost savings from acquiring more fixed resources.

The marginal analysis problem above is straightforward to solve in the context of a traditional queuing model using classic results by Erlang, assuming arriving jobs never wait for resources [38, 75, 86]. Variants of this classic problem have been addressed in prior work both generally, and in the context of cloud computing.

Marginal Analysis under Waiting. The classic marginal analysis above implicitly assumes jobs never wait for resources, and always immediately execute on either a reserved or on-demand resource. A key insight of our work is that cloud-enabled schedulers can explicitly control whether (and how long) jobs wait for reserved resources if they are busy, and that this waiting policy affects the optimal provisioning of reserved resources that minimizes cost. We know of no work that explicitly defines and analyzes such waiting policies for cloud-enabled schedulers by applying marginal analysis.

6.4 Non-selective Waiting Policies

We develop a simple queuing model for cloud-enabled schedulers to understand the relationship between the waiting policy, fixed resource provisioning, job waiting time, and cost. We first analyze basic non-selective waiting policiesAll Jobs Wait (AJW), No Jobs Wait (NJW), and All Jobs Wait Threshold (AJW-T)—which apply the same policy to all jobs.

Our analysis extends a $M/M/s/\infty$ queuing model using s fixed resources with first-come-first-serve (FCFS) scheduling, mean job arrival rate λ, and mean job service time $1/\mu$, where job arrivals follow a Poisson process, job service times are i.i.d. and exponentially distributed, and each resource executes one job at a time. The

offered load is $a{=}\lambda/\mu$, and the offered load (and utilization) per fixed resource is $\rho{=}a/s{=}\lambda/(s{\times}\mu)$.

6.4.1 All Jobs Wait (AJW)

Model Analysis. All Jobs Wait (AJW) is a baseline policy that requires all jobs to wait for fixed resources, and never rents on-demand resources. We present it as a foundation for our subsequent analysis. AJW's analysis is equivalent to that of an $M/M/s/\infty$ queue. The effective price P for each fixed resource is simply a function of the mean resource utilization ρ and fixed resource price p_f at full utilization, as shown below.

$$P = p_f/\rho \tag{6.1}$$

Thus, as mean utilization ρ increases, the effective price decreases up to 100% utilization. Of course, as utilization increases, the mean waiting time w in the queue also increases. The mean waiting time w for fixed resources under AJW is a well-known function, shown below, of s, λ, and μ, where $C(s,a){=}[(s \times a^s)/(s! \times (s - a))]/[\sum_{i=0}^{s-1} a^i/i! + (s \times a^s)/(s! \times (s - a))]$ is Erlang's delay (or C) formula.

$$w = \frac{C(s,a)}{s \times \mu - \lambda} \tag{6.2}$$

Empirical Validation. *We empirically validate the effective price P and mean waiting time w for all waiting policy models we present for the same baseline example.* In our baseline example, we set $\lambda{=}0.2$ (or 1 job every 5 seconds on average), $\mu{=}0.002$ (or an average job service time of 500 seconds), $p_o{=}9.6$¢/hour, and $p_f{=}3.84$¢/hour. Thus, in this case, the discount factor d for fixed resources at 100% utilization is $p_f/p_o{=}0.4$. We plot both the continuous function from our model, as well as average empirical values from 20 trials of our job simulator. Each trial simulates the model on a synthetically generated job trace with 2M jobs using exponentially distributed

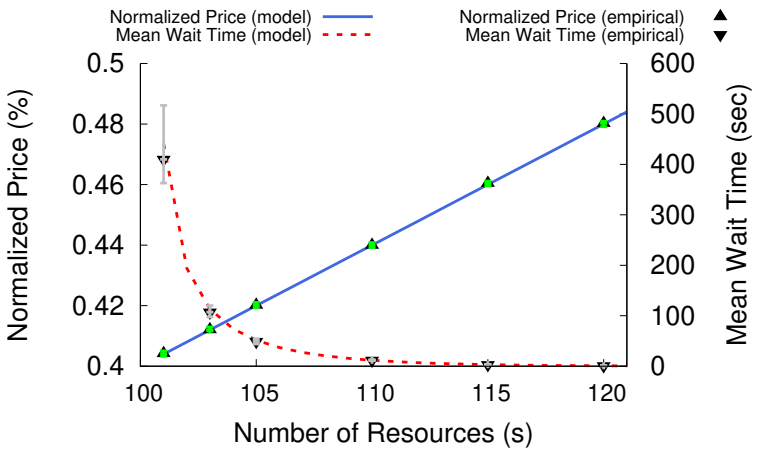


Figure 6.2: Normalized price P (left y-axis) and mean wait time w (right y-axis) as a function of fixed resources s under AJW.

inter-arrival and service times using the baseline parameters, as well as any model-specific parameters. To capture steady states, we do not include the first and last 10% of jobs when computing P and w.

For AJW, Figure 6.2 plots the effective price P (left y-axis) and the mean wait time W (right y-axis), obtained from our model and from simulations, as a function of the fixed resources s. Here, *as in all subsequent graphs*, we normalize the effective price P by the price of on-demand resources p_o. The minimum value on the left y-axis is $P{=}p_f{=}0.4$, since this represents the lowest possible price (when using only fixed resources at 100% utilization). Figure 6.2 shows that our model's predictions closely match the empirical results, both for the normalized price and the mean waiting time. Also, as expected, the graph shows that as s increases the effective price P increases linearly due to the decrease in mean utilization ρ. In contrast, the mean waiting time decreases super-linearly with increasing s. Thus, AJW offers a risky tradeoff between w and P, since provisioning fixed resources for high utilization, i.e., a low s, to reduce the price may cause high waiting times.

Key Point. *Since waiting time increases super-linearly as utilization* $\rho{\to}100\%$*, AJW encourages over-provisioning to ensure a utilization below 100% with waiting times near* 0.

6.4.2 No Jobs Wait

Model Analysis. The No Jobs Wait (NJW) waiting policy is similar to existing auto-scaling policies for cloud-enabled schedulers that execute jobs on fixed resources when available, and dynamically acquire on-demand resources to execute jobs when all fixed resources are busy. Recall from 6.3 that, given a workload, there is an optimal number of fixed resources s for NJW that minimizes cost, and this value occurs when the s^{th} resource has a utilization equal to the fixed resource's discount factor d. Thus, to optimize s under NJW, we need an expression for the s^{th} resource's utilization, denoted as ρ_s.

We find ρ_s using marginal analysis by applying Erlang's loss (or B) formula, which assumes a $M/M/s/0$ queue. To compute the utilization of the s^{th} resource, we first compute the difference between the blocking probability when using $s-1$ and when using s resources. This difference represents the percentage of jobs an additional resource serves. Multiplying this percentage by the offered load $a{=}\lambda/\mu$ gives the mean utilization of the s^{th} resource ρ_s, as shown below, where $B(s,a){=}(a^s/s!)/(\sum_{i=0}^{s}(a^i/i!))$ is Erlang's loss (or B) formula.

$$\rho_s = a \times [B(s-1,a) - B(s,a)] \tag{6.3}$$

Under a No Jobs Wait (NJW) waiting policy, rather than actually exit the system, the scheduler acquires on-demand resources to immediately service blocking jobs without waiting. To determine the optimal number of fixed resources s that minimizes cost, we set the discount factor d equal to ρ_s in Equation 6.3 and solve for s. Since Erlang's loss formula includes a factorial and summation, there is no closed-form expression for s, requiring us to solve for it numerically. Since ρ_s is monotonically decreasing as s increases, we can use a binary search to determine the optimal s.

$$P = (1-r) \times \frac{p_f}{\rho_f} + r \times p_o \tag{6.4}$$

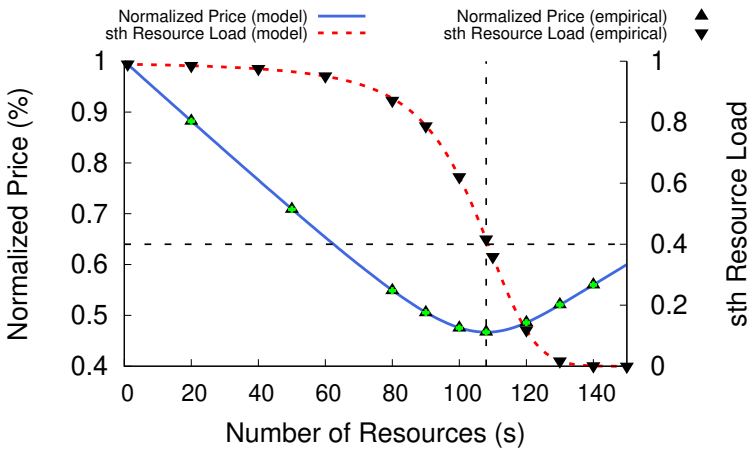
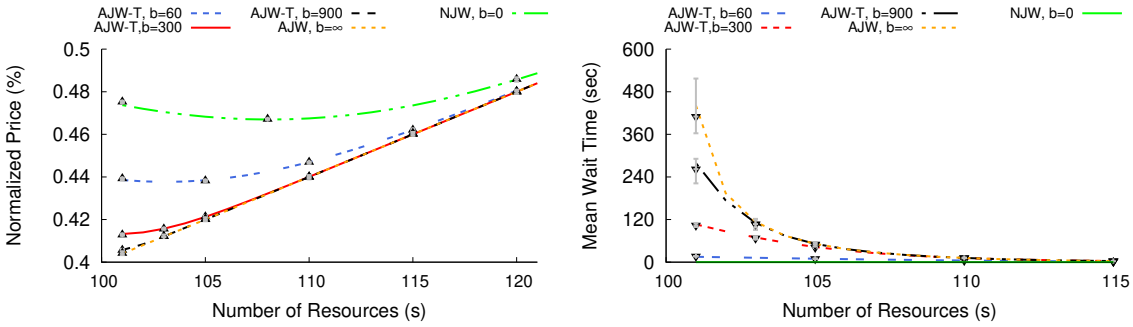


Figure 6.3: Normalized price P (left y-axis) and mean utilization of the sth resource ρ_s (right y-axis) as a function of fixed resources s under NJW. The minimum price occurs when the fixed resources' discount factor d=ρ_s.

Here, we use r to represent the fraction of the workload that executes on on-demand resources. The first additive term normalizes the price of the s fixed resources by their mean utilization ρ. The second additive term simply multiplies the price of on-demand resources p_o by the remaining fraction of the workload r. For NJW, r=$B(s, a)$, as this represents the probability that a job blocks and then runs on on-demand resources.

Empirical Validation. Figure 6.3 shows the effective price P (left y-axis) as a function of fixed resources s under NJW, where we again normalize P by the price of on-demand resources p_o. The right y-axis shows the mean utilization of the sth resource ρ_s, as the waiting time w is always zero under NJW. As expected, the graph shows the model closely matches the empirical results. As s increases, the effective price decreases to the optimal s=108 where ρ_s equals the 0.4 discount factor, after which, the effective price increases. At the optimal s=108, NJW has an effective price P=0.467×0.096=\$0.044832/hour, while AJW's price is ∼7.5% less at P=0.432×0.096=\$0.041472/hour. However, under NJW, jobs never incur waiting time, while AJW incurs a mean waiting time of 20s, with some jobs waiting much longer.



(a) Normalized price (b) Mean waiting time

Figure 6.4: Normalized price P (a) and Mean waiting time w (b) as a function of fixed resources s under AJW-T for different threshold waiting times b.

Key Point. *While NJW's cost is higher than AJW's for the same fixed resources, it guarantees no waiting time. NJW encourages optimal provisioning, since its cost increases as fixed resource provisioning deviates from the optimal.*

6.4.3 All Jobs Wait - Threshold

Model Analysis. AJW and NJW define two extremes: AJW yields a low price but with a potentially high waiting time, while NJW yields a higher price but zero waiting time. The All Jobs Wait-Threshold (AJW-T) waiting policy defines a continuous tradeoff between these two extremes by requiring all jobs to wait up to some threshold time b, at which point the scheduler acquires an on-demand resource to service them. At b=0, AJW-T is equivalent to NJW, and as $b \to \infty$, AJW-T approaches AJW. In queuing literature, AJW-T is equivalent to a queuing model with reneging jobs that exit the queue after waiting a threshold period. The reneging probability r is given by the following lemma, which follows from an analysis by Liu and Kulkarni [55].

Lemma 6.4.1 *The reneging probability r in a $M/M/s/\infty$ system is computed as follows.*

$$r = \frac{\alpha \cdot \beta \cdot e^{-\delta \cdot b}}{s \cdot \mu} \tag{6.5}$$

where

$$\delta = (s\mu - \lambda) \tag{6.6}$$

$$\beta = \frac{s\mu p}{1-p} \tag{6.7}$$

$$p = \frac{(\lambda/\mu)^s}{s! \sum_{i=0}^{s} \frac{(\lambda/\mu)^i}{i!}} \tag{6.8}$$

$$\alpha = \begin{cases} [\beta(\frac{1}{\delta} - e^{\delta \cdot b} \cdot \frac{\lambda}{\delta \cdot s\mu}) + 1]^{-1} & \rho \neq 1 \\ \frac{\lambda}{\lambda + \beta \cdot (\lambda \cdot b + 1)} & \rho = 1 \end{cases} \tag{6.9}$$

As before, we need an expression for the mean utilization of the sth resource, as in Equation 6.3, to solve for the optimal s that minimizes cost. However, in this case, we replace Erlang's B formula with r above when using $s-1$ and s resources, as shown below, since r represents the reneging probability under AJW-T, which is akin to the blocking probability under AJW. We can again solve for the optimal s that minimizes price numerically using a binary search, as ρ_s is still monotonically decreasing as s increases, where a=λ/μ.

$$\rho_s = a \times [r_{s-1} - r_s] \tag{6.10}$$

After determining the optimal s and r for a given threshold waiting time b, we compute the mean waiting time of jobs. Liu and Kulkarni give the mean waiting time under reneging as follows [55]. The first additive term represents the mean waiting time for the jobs that execute on fixed resources, while the second additive term represents the mean waiting time for jobs that execute on on-demand resources, which is simply $r \times b$ as they all wait the maximum time b.

$$w = \begin{cases} (1-r) \times (\frac{\alpha \times \beta (1 - \delta b e^{-\delta \times b} - e^{-\delta \times b})}{(1-r) \times \delta^2}) + r \times b & \rho \neq 1 \\ (1-r) \times (\frac{\alpha \times \beta \times b^2}{(1-r) \times 2}) + r \times b & \rho = 1 \end{cases} \tag{6.11}$$

Empirical Validation. Figure 6.4 shows the effective price P (a) and mean waiting time w (b) as a function of fixed resources s under AJW-T for different threshold maximum waiting times b, as well as the price under AJW and NJW. Once again, the model's predictions closely match the empirical results. As expected, as b increases, the price approaches AJW, and as it decrease the price approaches NJW. The graph also shows that as b increases, the optimal fixed resources s that minimizes price decreases. Figure 6.4 (b) shows that as b increases, the mean waiting time increases more sharply as $s \rightarrow 100$. Thus, unlike AJW and NJW, AJW-T is configurable, enabling users to set their own tradeoff between price and waiting time.

Key Point. *AJW-T offers a configurable tradeoff between price and waiting time by enabling users to set the maximum waiting time threshold b, unlike NJW, which offers no tradeoff, and AJW, which offers a risky tradeoff.*

6.5 Selective Waiting Policies

Unlike non-selective waiting policies, selective waiting policies do not apply to all jobs, but only to selected jobs based on system or job characteristics. We define and analyze two selective policies: Short Waits Wait (SWW) and Long Jobs Wait (LJW). Since waiting policies are not mutually exclusive, we also analyze a compound waiting policy that combines SWW, LJW, and the threshold waiting time from AJW-T.

6.5.1 Short Waits Wait

Model Analysis. Unlike AJW-T where jobs wait up to a threshold value before they are scheduled on on-demand resources, in the Short Waits Wait (SWW) waiting policy, incoming jobs *estimate* their waiting time upon arrival (based on the jobs

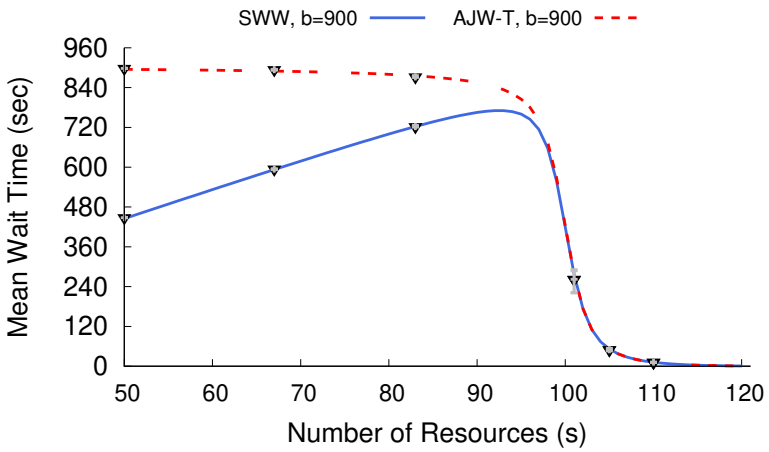


Figure 6.5: Mean waiting time as a function of fixed resources under SWW and AJW-T where b=900s=15m.

running and ahead of it in the queue) and only wait if the estimated wait time is short, i.e., less than a threshold value. If the estimated wait time is long, i.e, exceeds the threshold, then they immediately run on on-demand resources without waiting. In queuing literature, this behavior is equivalent to a queuing system with balking jobs. Importantly, as prior work shows, the same set of jobs that renege under AJW-T, and in our case run on on-demand resources, will also balk under SWW [55]. Thus, the fraction of jobs r that run on on-demand resources under SWW is the same as under AJW-T (from Lemma 6.4.1), and thus the effective price for resources is the same under AJW-T and SWW for the same b.

The only change with SWW relative to AJW-T is the mean waiting time w, since under SWW jobs exit the system immediately and run on on-demand resources if their waiting time *would* exceed the threshold waiting time b. In this case, the mean waiting time w shown below is the same as in Equation 6.11 except that we remove the $r \times b$ term, since the r fraction of jobs that run on on-demand resources incur no waiting time rather than incurring b waiting time, as in AJW-T.

$$w = \begin{cases} (1-r) \times \left(\frac{\alpha \times \beta (1 - \delta b e^{-\delta \times b} - e^{-\delta \times b})}{(1-r) \times \delta^2}\right) & \rho \neq 1 \\ (1-r) \times \left(\frac{\alpha \times \beta \times b^2}{(1-r) \times 2}\right) & \rho = 1 \end{cases} \tag{6.12}$$

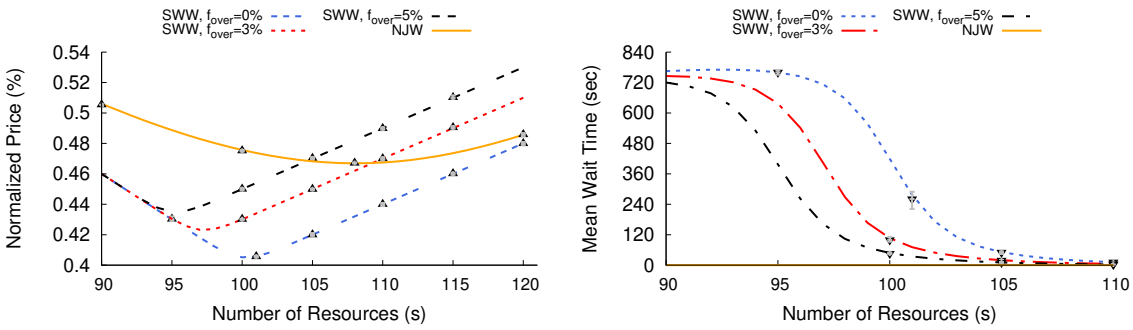
Empirical Validation. Figure 6.5 plots the mean waiting time w for SWW and AJW-T as a function of the fixed resources s, and a threshold waiting time b=900s=15m. The mean waiting time for SWW approaches zero as s decreases (and load increases) rather than b for AJW-T, as increasingly more jobs exit the system without waiting and run on on-demand resources. Note that SWW's mean waiting time reaches its maximum at s=93, and is always less than that of AJW-T.

Key Point. *SWW is strictly better than AJW-T for the same threshold b, yielding same price at a lower mean waiting time.*

6.5.1.1 Prediction Accuracy

The SWW analysis above assumes that arriving jobs are able to perfectly predict their waiting time w. Doing so requires perfectly predicting the running time of every job currently running and ahead of them in the queue. There is significant prior work on predicting queue waiting times [28, 72] using statistical analyses and machine learning classifiers. This prior work demonstrates that accurately predicting queue waiting times can be challenging. As a result, we also model and analyze SWW under inaccurate predictions of job waiting time. Importantly, the goal of our work is not to develop a better waiting time predictor, but to reason about the effectiveness of prediction methods

Given a threshold waiting time b, there are two misprediction cases to consider: the scheduler either i) over-predicts a job's waiting time and thus runs it on on-demand resources when it should have waited for fixed resources, or ii) under-predicts a job's waiting time and thus forces it to wait for fixed resources when it should have run immediately on on-demand resources. We consider each case separately based on the fraction of jobs f_{over} and f_{under} that over- and under-predict their waiting time, respectively.



(a) Normalized price (b) Mean waiting time

Figure 6.6: Normalized price P (a) and Mean waiting time w (b) as a function of s under SWW for different over-prediction errors f_{over} and NJW.

Over-predicting Waiting Time. As the fraction of jobs that over-predict waiting time approaches 100%, SWW approaches the behavior of using all on-demand resources (plus the cost of the fixed resources), as jobs always immediately exit the system (due to their high predicted waiting time) and run on on-demand resources. For simplicity, our analysis here is not work-conserving, such that over-predictions redirect jobs to on-demand resources even when fixed resources are available.

$$P = (1 - r) \times \frac{p_f}{\rho_f} + (1 - f_{over}) \times r \times p_o + f_{over} \times p_o \tag{6.13}$$

Figure 6.6 (a) shows the effective price P (normalized by the on-demand price as before) on the y-axis as a function of s. As the graph shows, as f_{over} increases to one, the optimal value of s changes, and approaches that under NJW. Note that the price of a work-conserving variant would be bounded by NJW as s increases, rather than exceeding it, since it would utilize any idle fixed resources. Similarly, Figure 6.6 (b) shows the mean waiting time w as a function of s. As expected, as f_{over} increases, the mean waiting time decreases (as fewer jobs wait).

Key Point. *SWW is sensitive to over-predictions, as 3-5% over-predictions significantly alters the price and mean waiting time.*

Under-predicting Waiting Time. As the fraction of jobs f_{under} that under-predict their waiting time approaches 100%, SWW approaches the behavior of AJW-T, since

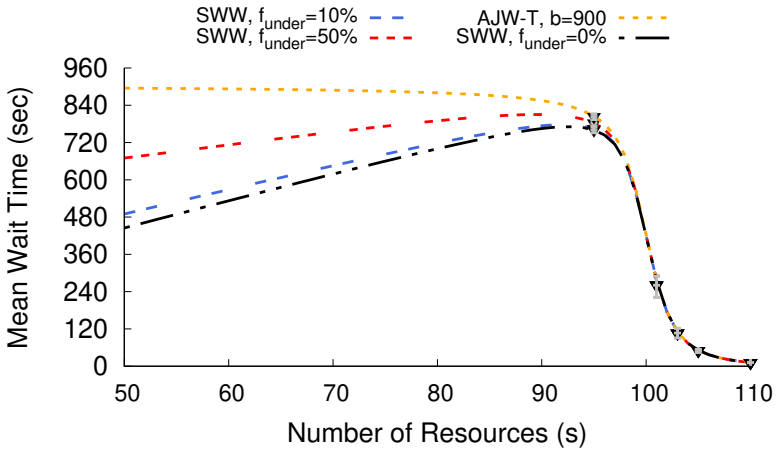


Figure 6.7: Mean waiting time as a function of fixed resources s under SWW for different under prediction rates f_{under}.

jobs always wait for fixed resources up to threshold b before running on on-demand resources. The effective price under SWW with under-predictions is the same as that with AJW-T and SWW with perfect predictions, as the same set of jobs run on on-demand resources in all cases. The only difference is the job waiting times.

Figure 6.7 shows the mean waiting time w as a function of s using our baseline parameters for different values of f_{under}, as well as for AJW-T with b=900s. As expected, as f_{under} increases, the mean waiting time increases until it matches that of AJW-T.

Key Point. *SWW is not highly sensitive to under-predictions, as they do not affect the effective price and only affect the mean waiting time when fixed resources are under-provisioned.*

Our results are important in assessing and contextualizing the accuracy of new and existing methods for predicting queue waiting times. Specifically, for cloud-enabled schedulers, these prediction techniques should focus on minimizing over-predictions.

6.5.2 Long Jobs Wait

Model Analysis. Long Jobs Wait's (LJW) intuition is that longer running jobs should be willing to wait longer for fixed resources, since longer waiting times are a

smaller percentage of their overall running time compared to shorter jobs. For LJW, we introduce a running time threshold t such that jobs shorter than t run immediately on on-demand resources, while others wait for fixed resources. For simplicity, our LJW policy is not work-conserving in that it runs short jobs on on-demand resources even if fixed resources are available. This non-work-conserving variant will behave similarly to a work-conserving one in the optimal case when fixed resources are not over-provisioned (and thus rarely idle). For LJW, we separate the analysis for short jobs and long jobs. As shown below, the effective price P is the weighted average of the price to run short and long jobs. As before, r represents the fraction of jobs that run on on-demand resources, while P_{short} and P_{long} represent the price to run short and long jobs, and μ_{short} and μ_{long} represent the mean service rate of short and long jobs.

$$P = (1 - r) \times \frac{\mu}{\mu_{long}} \times P_{long} + r \times \frac{\mu}{\mu_{short}} \times P_{short} \tag{6.14}$$

Thus, first and second additive terms represent the relative cost to execute long and short jobs, respectively, based on their fraction of the total jobs, their proportion of the service time, and their price. Note that, $\mu_{long} > \mu > \mu_{short}$ for any $t > 0$. Similarly, the mean waiting time w is the weighted average of the waiting time to run short and long jobs. Since, by definition, short jobs do not wait, w is only dependent on the fraction of long jobs and their mean waiting time.

$$w = (1 - r) \times w_{long} \tag{6.15}$$

Short Jobs. All short jobs (with running times $<t$) run on on-demand resources at price p_o without any waiting time. Thus, P_{short}=p_o, while r is the fraction of jobs with running times less than t, which is equivalent to the CDF of the exponential distribution for service times at x=t, as shown below.

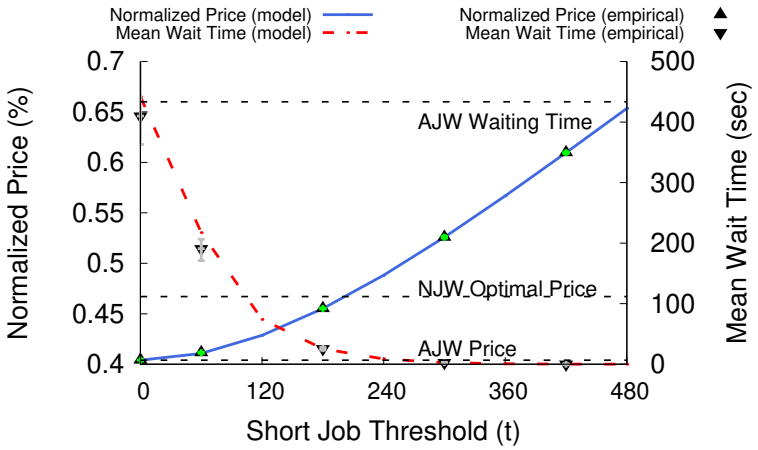


Figure 6.8: Normalized price P and mean wait time w as a function of the short job threshold t (in seconds) for s=101 under an LJW waiting policy.

$$r = 1 - e^{-\mu t} \tag{6.16}$$

Long Jobs. Since long jobs always wait for fixed resources, the policy is similar to AJW in 6.4.1 but applied to long jobs. The mean arrival rate for long jobs is λ_{long} is the product of the overall job arrival rate λ and the fraction of long jobs $(1 - r)$.

$$\lambda_{long} = \lambda \times (1 - r) = \lambda \times e^{-\mu t} \tag{6.17}$$

Similarly, we compute the mean service rate μ_{long} for long jobs using its service time PDF $f(x, \mu)$, as below. The PDF for long jobs is an exponential distribution shifted by t units.

$$f(x, \mu) = \mu e^{-\mu(x-t)}, x \geq t \tag{6.18}$$

We find the expected value of the long jobs service time PDF to derive its mean service time $\frac{1}{\mu_{long}}$ by integrating from x=t→∞.

$$\frac{1}{\mu_{long}} = \int_t^{\infty} x\mu e^{-\mu(x-t)} dx = t + \frac{1}{\mu} \tag{6.19}$$

Note that we can derive μ_{short} from μ_{long}, r, and μ, since the mean service time of the original distribution $1/\mu$ is the weighted average of the mean service time of

short jobs $1/\mu_{short}$ and long jobs $1/\mu_{long}$. Thus, we compute μ_{short} by simply solving the expression below.

$$\frac{1}{\mu} = r \times \frac{1}{\mu_{short}} + (1-r) \times \frac{1}{\mu_{long}} \tag{6.20}$$

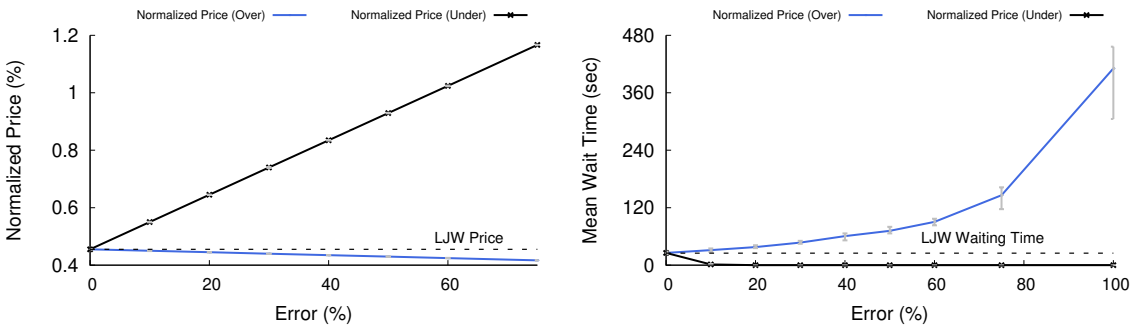
The effective price P_{long} of running long jobs on fixed resources is simply the price of fixed resources p_f at full utilization divided by the actual utilization ρ_{long}, where $\rho_{long} = \lambda_{long}/(s \times \mu_{long})$.

$$P_{long} = \frac{p_f}{\rho_f} = \frac{p_f \times s \times \mu_{long}}{\lambda_{long}} \tag{6.21}$$

Importantly, however, the distribution of jobs with service times greater than t is *not* exponentially distributed. As a result, we *cannot* apply the same model as for AJW to compute the waiting time. Instead, we use the well-known approximation below for the waiting time of an M/G/s queue, where CV is the distribution's coefficient of variation, i.e., the standard deviation divided by the mean. In this case, the standard deviation of the long jobs' service time distribution is $1/\mu$, and the mean is $1/\mu_{long}$, so $CV{=}\mu_{long}/\mu$.

$$w \sim \frac{CV^2+1}{2} \times \frac{C(s,a)}{s \times \mu_{long} - \lambda_{long}} \tag{6.22}$$

Empirical Validation. Figure 6.8 shows the normalized price (left y-axis) and waiting time (right y-axis) under LJW as a function of t for $s{=}101$, as well as AJW and NJW, using our baseline parameters. As before, the graph shows that the empirical values closely match the model's waiting time approximation above. The graph shows that as t increases the normalized price increases, as fewer jobs wait for resources. However, LJW also significantly decreases the mean waiting time relative to AJW as t increases, since the exponential service time distribution is weighted towards short jobs, which experience no waiting time under LJW. In addition, since long jobs still comprise a high fraction of the overall service time (and thus cost), the effective price



(a) Normalized Price (b) Mean Job Waiting Time

Figure 6.9: Normalized price (a) and mean job waiting time (b) as a function of the fraction of jobs with incorrect over- and under-predictions (%) of job running time for s=101 and t=180 under an LJW waiting policy.

under LJW, especially for small values of t, increases at a much lower rate than the waiting time decreases.

Key Point. *LJW offers a nice tradeoff: as t increases, price increases modestly, while waiting time decreases significantly.*

6.5.2.1 Prediction Accuracy

Our LJW analysis above assumes that arriving jobs perfectly predict their running time, which may not always be possible. As with predictions of queue waiting time, there is significant prior work [59, 77] on predicting job running time, since it is an important input for many common scheduling policies, such as SJF. As in §6.5.1.1, our analysis provides a basis for contextualizing this prior work, and understanding how inaccuracy can affect waiting policies. At a high level, similar to SWW's analysis, as f_{over}—the fraction of short jobs that are predicted to be long (with running times $>t$)—approaches one, LJW approaches the behavior of AJW, since all jobs wait. In contrast, as f_{under}—the fraction of long jobs that are predicted to be short—approaches one, LJW approaches using all on-demand resources (plus the cost of fixed resources).

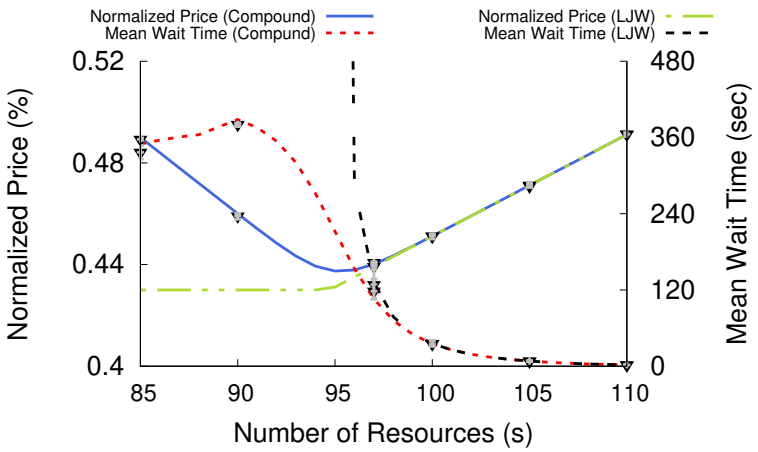


Figure 6.10: Normalized price P and mean wait time w as a function of fixed resources s for our compound policy (b=900 and t=180) and LJW (t=180).

To understand how sensitive LJW is to over- and under-predictions of job running time, we plot the normalized price and mean waiting time as a function of f_{under} and f_{over} for s=101 and t=180. We only plot empirical results from our job simulator, since we have no analytical model. Figure 6.9(a) shows that as the over-prediction rate increases, the effective price decreases, but, since LJW's price in this case is already near the optimal price p_f, the decrease is minimal. In contrast, as the under-prediction rate increases, the effective price increases significantly. Figure 6.9(b) shows the opposite effect: as the over-prediction rate increases, the mean waiting time increases significantly, while as the under-prediction rate increases the mean waiting time decreases, although since LJW's mean wait time is already near zero, the decrease is not significant.

Key Point. *LJW's effective price is robust to over-predictions and sensitive to under-predictions, while its mean waiting time is robust to under-predictions and sensitive to over-predictions. LJW is more sensitive to over-predictions, since they cause a super-linear increase in waiting time for only a linear decrease in price.*

6.5.3 Compound Waiting Policies

Model Analysis. Waiting policies, unlike scheduling policies, are not mutually exclusive. That is, we can concurrently apply multiple waiting policies that select jobs to wait based on different characteristics. Thus, we analyze a compound waiting policy that combines the advantages of AJW-T, SWW, and LJW. In analyzing this policy, we first apply LJW's analysis from 6.5.2, since its waiting decisions are based on job running time, and are thus load insensitive and not affected by other waiting policies. Our LJW analysis yields a fraction r of short jobs that always run on on-demand resources, which we label r_{short}. The remaining $(1-r_{short})$ long jobs run on fixed or on-demand resources depending on their waiting time.

We next apply SWW's analysis from 6.5.1 solely to the remaining long jobs. In particular, we compute the fraction r_{sww} of the remaining long jobs that run on on-demand resources (due to long wait times) by applying Lemma 6.4.1 using λ_{long} and μ_{long} from 6.5.2 for a given value of s and b. This is an approximation, since Lemma 6.4.1 assumes exponentially distributed service times, and the long jobs' service time distribution is an exponential distribution truncated at t. This approximation becomes more accurate as $t \to 0$ and the distribution approaches an exponential. Given r_{sww}, the effective price for our compound waiting policy is as follows.

$$P = (1 - r_{short}) \times (1 - r_{sww}) \times \frac{\mu}{\mu_{long}} \times \frac{p_f}{\rho_f} + (1 - r_{short}) \times r_{sww} \times \frac{\mu}{\mu_{long}} \times p_o + r_{short} \times \frac{\mu}{\mu_{short}} \times p_o \quad (6.23)$$

Here, ρ_f, shown below, is the mean utilization of the fixed resources, which is simply the adjusted arrival rate of jobs to the fixed resources divided by their mean service rate, and then normalized by s.

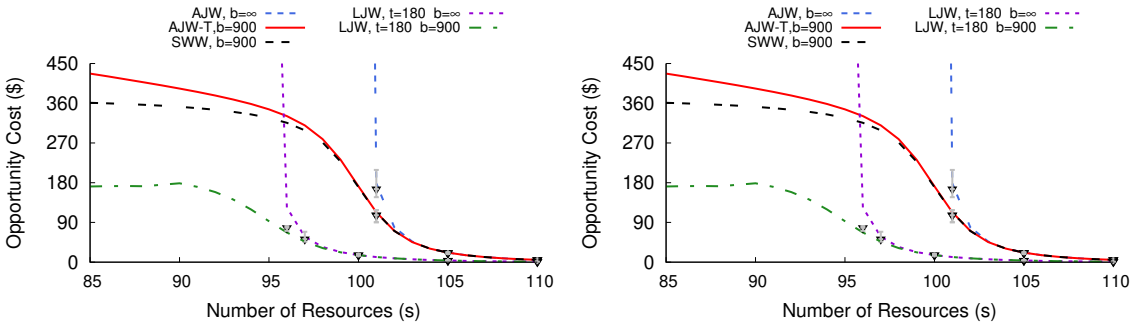


Figure 6.11: Opportunity cost as a function of fixed resources s under AJW, AJW-T, SWW, LJW, and compound policy when using (a) *FCFS scheduling* and (b) *SJF scheduling*

$$\rho_f = \frac{(1 - r_{short}) \times (1 - r_{sww}) \times \lambda}{s \times \mu_{long}} \tag{6.24}$$

We use the same approach as in LJW to approximate the compound policy's mean waiting time, but replace the waiting time under AJW with the waiting time under SWW from Equation 6.12 as below, again using λ_{long} and μ_{long} as the input. The coefficient of variation CV is the same as in LJW.

$$w \sim \begin{cases} \frac{CV^2+1}{2} \times (1 - r_{sww}) \times \left(\frac{\alpha \times \beta (1 - \delta b e^{-\delta \times b} - e^{-\delta \times b})}{(1 - r_{sww}) \times \delta^2}\right) & \rho < 1 \\ \frac{CV^2+1}{2} \times (1 - r_{sww}) \times \left(\frac{\alpha \times \beta \times b^2}{(1 - r_{sww}) \times 2}\right) & \rho = 1 \end{cases} \tag{6.25}$$

Empirical Validation. Figure 6.10 compares our compound waiting policy with LJW using our baseline parameters with b=900 and t=180. The primary advantage of the compound policy over LJW is that it strictly lowers the overall waiting time, since long jobs do not wait indefinitely, which is especially important when resources are under-provisioned, for nearly the same effective price. As shown, the compound policy's mean waiting is less than or equal to that of the LJW policy.

Key Point. *Our compound policy combines the advantages of AJW-T, SWW, and LJW, and thus offers the best tradeoff.*

6.5.4 Model Results Summary

Our analyses show that waiting policies offer a complex tradeoff between fixed resource provisioning, cost, and waiting time. To summarize these tradeoffs, we define a new metric, called the *opportunity cost of waiting*, which values a job's waiting time equal to its running time. The mean opportunity cost $P \times w$ and is in dollars, where lower values of P and w are better. Figure 6.11 (a) shows the mean opportunity cost of waiting for AJW, AJW-T (for b=900), SWW (for b=900), LJW (for t=180), and our compound policy (for b=900 and t=180) using our baseline parameters. We exclude NJW, as its opportunity cost is always zero, since its waiting time is zero. As shown, for the remaining policies where a price-waiting time tradeoff exists, our compound policy yields the lowest opportunity cost.

In addition, the general insights above also hold for different scheduling policies. While the waiting policy is distinct from the scheduling policy, and both can be defined independently, there is some interaction between them. Figure 6.11 (b) shows the same experiment as Figure 6.11 (a), but with shortest job first (SJF) as the scheduling policy instead of FCFS. The graph shows that the relative ordering of waiting policies is the same when using SJF and FCFS, and also that the trends are the same. Of course, the absolute opportunity cost when using SJF is significantly less because SJF substantially decreases the waiting time for jobs that wait for fixed resources.

6.6 Implementation

We implemented a waiting policy model analyzer based on our analysis, as well as a trace-driven job simulator, in python.

Model Analyzer. Our model analyzer implements the analytical queuing model for all the waiting policies we analyze. The analyzer enables what-if analyses to compare and understand a workload's expected cost and job waiting times under different

policies and parameter values. The analyzer takes as input a policy's name and λ, μ, s, p_f, and p_o, as well as b for AJW-T, SWW, and the compound policy, and t for LJW and the compound policy. The analyzer's output is the policy's mean waiting time w, the effective price P, and the fraction of jobs that run on on-demand resources r.

Job Simulator. We implemented a trace-driven job simulator in python that mimics a cloud-enabled job scheduler, which can acquire VMs on-demand to service jobs. The simulator uses a FCFS scheduling policy, and also implements each of our waiting policies. The simulator takes as input a trace of jobs, s, p_f, the name of a waiting policy, and the same waiting policy-specific parameters as above. Users must also specify the number of cores and memory allotment for each fixed resource s. The simulator's output is the mean waiting time w, the effective price P, the fraction of jobs that run on on-demand resources r, and the total cost C.

Real-world Data. In evaluation, we use our job simulator to quantify the impact of different waiting policies on a real year-long job trace that includes 14M jobs from a production high-performance computing cluster consisting of 14.3k cores. The cluster is the University of Massachusetts (UMass) System Shared Cluster, and is available for general use to researchers from all five campuses in the UMass system, including its medical school [11]. Thus, the workload is a representative sample of job types across the entire scientific, engineering, and medical research communities. Each job entry in the log includes its submission time, user ID, maximum running time limit, requested number of cores and memory, and running time.

6.7 Evaluation

We do not intend our models to be predictive, but instead evaluate their usefulness in analyzing a real year-long batch workload. Specifically, we show that our models both 1) accurately predict the relative price and waiting time between different wait-

(a) Job Interarrival Times (b) Job Service Times (c) Job Resource Requirements

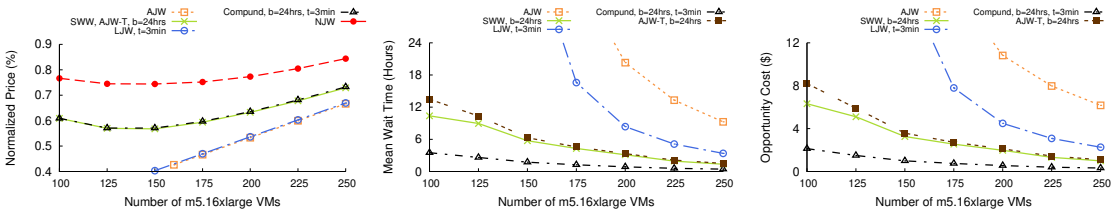
Figure 6.12: Histograms of job inter-arrival times (a) and service times (b) for our real production batch workload along with an exponential distribution using the same mean, as well as the mix of long and short jobs (c).

ing policies in our real workload, and 2) enable reasoning about price and waiting time by understanding the differences between our model's and the real workload.

Workload. Figure 6.12 characterizes our real workload and our model's ideal. Figure 6.12(a) is a histogram of job inter-arrival times for our trace and an exponential distribution with the same mean, which is 0.4527 jobs/sec. Note that the bin size is non-uniform, since our trace much more bursty than our model assumes. In particular, nearly 90% of job inter-arrival times are between 0 and 1 second compared with less than 40% for an exponential distribution with the same mean. Both distributions have a heavy tail with our job trace experiencing a few more extremely long inter-arrival times, between 3 minutes and 50 hours. Figure 6.12(b) is a similar histogram of job service times with a mean service time $1/\mu$ of 6225 seconds (or 1.73 hours) per job. Overall, the job service times in our trace have both a heavier head and tail compared to the exponential distribution. To further illustrate, Figure 6.12(c) shows that short jobs are a high fraction of jobs, even for large short job thresholds, but account for only a small fraction of the resource usage.

6.7.1 Real-world Workload Results

Our simulations assume a large `m5.16xlarge` VM with 64 cores and 256GB memory to mitigate imperfect job packing on VMs. We contextualize our results by comparing against the current fixed-size cluster, which consists of 14,376 cores and



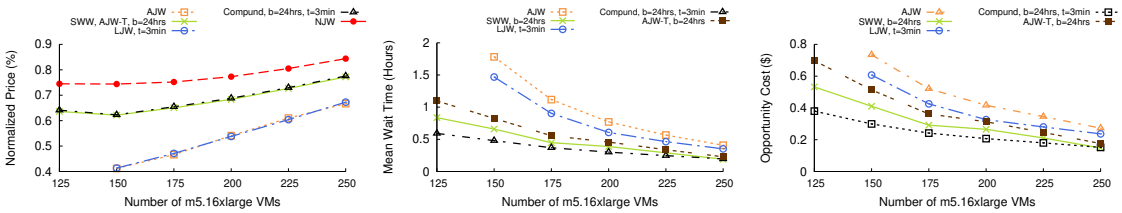
(a) Normalized Price (b) Mean Job Waiting Time (c) Opportunity Cost

Figure 6.13: Normalized price (a), mean job waiting time (b), and opportunity cost (c) as a function of `m5.16xlarge` VMs when executing our real production batch workload under AJW, AJW-T, SWW, LJW, and our compound policy with *FCFS scheduling policy*.

is equivalent to 225 `m5.16xlarge` VMs. Figure 6.13 shows the normalized price (a), mean waiting time (b), and opportunity cost (c) for each of our waiting policies. We select the maximum waiting time threshold b=24 hours for SWW and AJW-T, or slightly less than double the current cluster's mean waiting time using AJW. We select the long job cutoff t=3m where 60% of jobs are short and 40% are long.

Price. As expected, Figure 6.13(a) shows that AJW yields the lowest price, since it requires all jobs to wait for fixed resources. Interestingly, LJW yields nearly the same price even though it executes 60% of the total jobs on on-demand VMs. Since these 60% of short jobs comprise only a small fraction of the overall job execution time, executing them on on-demand VMs does not substantially increase the normalized price. SWW, AJW-T, and our compound policy yield nearly the same price for the same reason. When using AJW, our current cluster yields a normalized price of 0.6 at x=225 fixed resources, while the minimum cost under the compound policy is 0.571 at x=150, or 5% less.

Waiting Time. As our model predicts, Figure 6.13(b) shows that the mean job waiting time under AJW and LJW increases super-linearly as fixed resources decrease. However, even though LJW's cost is nearly the same as AJW's, its mean waiting time is substantially less because the large fraction of short jobs never wait. In contrast, the mean waiting time under AJW-T, SWW, and the compound policy increases



(a) Normalized Price (b) Mean Job Waiting Time (c) Opportunity Cost

Figure 6.14: Normalized price (a), mean job waiting time (b), and opportunity cost (c) as a function of `m5.16xlarge` VMs when executing our real production batch workload under AJW, AJW-T, SWW, LJW, and our compound policy with *SJF scheduling policy*.

modestly as fixed resources decrease. Even at x=100, the mean waiting time of these policies is less than the 13.3 hour mean waiting time in our current fixed size cluster (AJW at x=225). At x=150, the compound policy has a mean waiting time of 1.74 hours, or 7× less than our current cluster (for 5% less cost).

Opportunity Cost. Figure 6.13(c) graphs the mean opportunity cost of waiting $P \times w$ for each policy, and shows that, as our model predicts, the compound policy offers the best tradeoff by a significant margin compared to the other policies. Note that, even though our workload's characteristics differ significantly from those assumed by our model, the overall trends in opportunity cost match those from our model in Figure 6.11.

Key Result. *At the optimal, the compound policy decreases the cost (by 5%) and mean job waiting time (by 7×) compared to the current cluster using AJW, and decreases the cost (by 43%) compared to renting on-demand resources for a comparatively modest increase in mean job waiting time (at* 1.74 *hours).*

SJF Scheduling. We next repeat the experiments above using the same parameters but using the SJF scheduling policy instead of FCFS scheduling. Figure 6.14 shows the results. As mentioned in §6.5.4, Figure 6.14(a) shows nearly the same normalized price across all the waiting policies as in Figure 6.13(a). In some cases, as with AJW and LJW, the price is the same, since these waiting policies are not sensitive to the

(a) Normalized Price (b) Mean Job Waiting Time (c) Opportunity Cost

Figure 6.15: Normalized price (a), mean job waiting time (b), and opportunity cost (c) as a function of the long job prediction error when executing our real production batch workload under a compound policy assuming 150 `m5.16xlarge VMs`.

scheduling policy. SWW is sensitive to the scheduling policy, and prioritizes short jobs on fixed resources, since these jobs have a lower waiting with SJF. However, since the vast majority of jobs in our real-world trace are already short, this only slightly increases the normalized price. Since the compound policy includes SWW, there is a similar impact on the normalized price.

Figure 6.14(b) shows that SJF significantly decreases the waiting time across all waiting policies compared to Figure 6.13(b).SJF is well-known to optimize for waiting time, often at the expense of starving longer jobs. However, in our case, long jobs never starve when using AJW-T, SWW, or the Compound policy. Importantly, the trends and relative ordering of the waiting policies under SJF is the same as under FCFS based on our analysis from §6.5.4.

Finally, Figure 6.14(c) shows the opportunity cost of all waiting policies under SJF. As with our model's workload in §6.5.4, the opportunity cost decreases compared to FCFS due to the substantial decrease in waiting time. As when using FCFS, the relative ordering of opportunity cost when using SFJ remains the same with the compound policy yielding the lowest opportunity cost.

6.7.2 Sensitivity Analysis

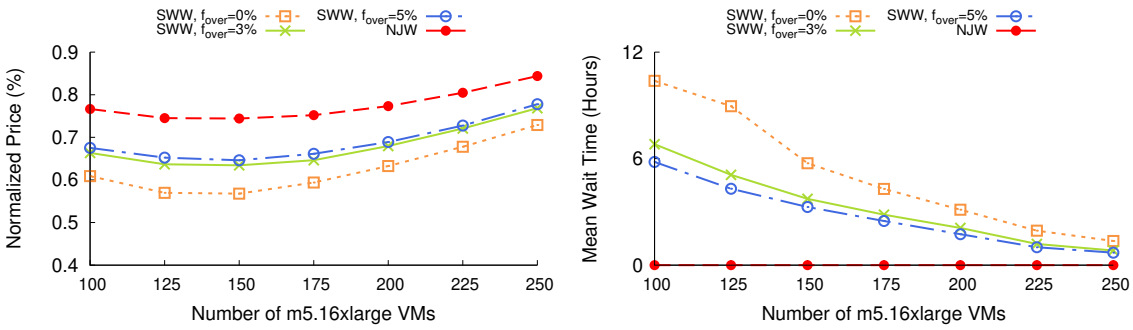
We perform a sensitivity analysis that varies errors in estimating job waiting time and running time to understand their effect on the results. We chose the values above

for b=24h and t=3m arbitrarily to be reasonable, as 24h is roughly twice the mean waiting time under AJW and t=3m categorizes a large fraction (60%) of jobs as short. We also assume accurate estimates of job waiting and running time, e.g., using historical data. Our sensitivity analysis assumes 150 `m5.16xlarge`'s when using the compound policy, as noted above.

Error Sensitivity. Figure 6.15 plots price, waiting time, and opportunity cost as a function of the short/long job prediction error, which is both the percentage of long jobs we mispredict as short, and short jobs we mispredict as long. Similarly, each line captures the waiting time threshold error, which is both the percentage of jobs that should wait but do not, and that do not but should. The graph shows price (a) is directly proportional to the short/long job prediction error, such that a 1% increase in error causes a 1% increase in price. In contrast, waiting time threshold errors are non-linear, with progressively lower price increases for each 10% increase in error. The graph still shows large savings compared to using on-demand even under high error rates. The mean waiting time (b) is much less affected by the short/long job prediction error, since a similar number of jobs must still wait (it is just the long jobs not waiting that increases the price). Finally, as above, the waiting time trend dominates the opportunity cost (c), and thus shows a similar trend.

6.7.3 Effect of Prediction Accuracy

To understand the effect of prediction accuracy for our waiting policies, we vary the errors in estimating job waiting time and running time in terms of their over- and under-predictions as in our analysis from §6.5.1.1 and §6.5.2.1. We use the baseline values of b=24h for the waiting time threshold and t=3m for the long job threshold. As in our model analysis, we consider the case of over predictions and under predictions separately based on the fraction of jobs f_{over} and f_{under} that over- and under-predict their job waiting time and job running time for SWW and LJW,



(a) Normalized Price (b) Mean Job Waiting Time

Figure 6.16: Normalized price (a) and mean job waiting time (b) as a function of fixed resources s when executing our real production batch workload under SWW for different over-prediction errors f_{over} and NJW.

respectively. In particular, we simulating each waiting policy using our job trace, we explicitly control the percentage of jobs that over- and under-predict waiting times and running times.

Over-predicting Waiting Time. Figure 6.16 plots the normalized price P and mean job waiting time W as a function of fixed resources s for different prediction errors f_{over} under SWW, where f_{over} is fraction of the jobs over-predicting their waiting time and thus runs it on on-demand resources when it should have waited for fixed resources. These graphs mirrors Figure 6.6 from §6.5.1.1 that uses our analytical model to quantify the effect of over-predictions of waiting time on the mean wait time and price. As the graph shows, the normalized price (a) increases with the over-prediction error. Figure 6.16(b) shows that the mean job waiting time decreases as f_{over} increases and eventually approaches 0 (or the behavior of NJW). As in our model, the mean wait time is monotonically decreasing and approaches 0 as the number of fixed resources increases. Importantly, our empirical results on over-predictions reinforce the key points from our models: that SWW is highly sensitive to over-predictions.

Under-predicting Waiting Time. Figure 6.17 plots the mean job waiting time w as a function of fixed resources s over different prediction errors f_{under} using SWW,

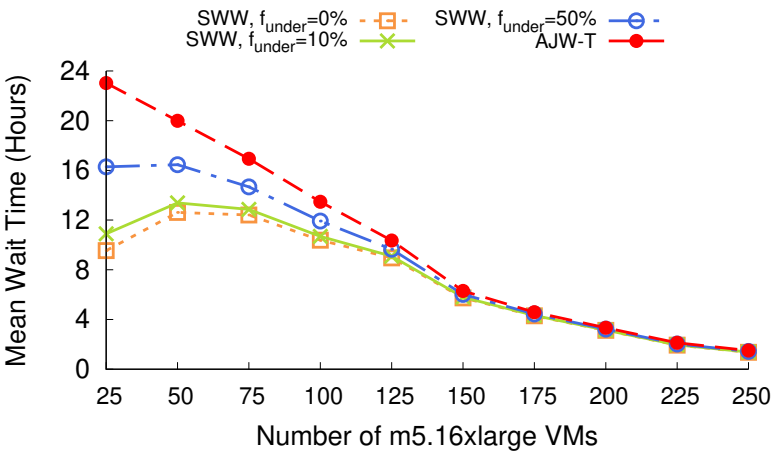
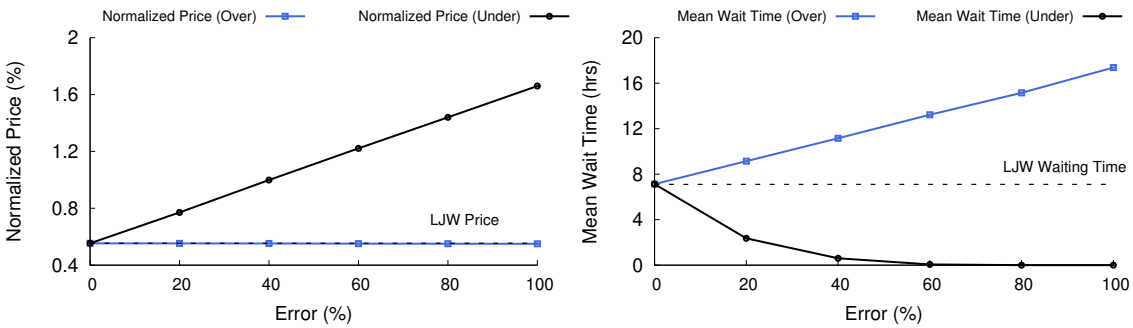


Figure 6.17: Mean waiting time as a function of fixed resources s when executing our real production batch workload under SWW for different over-prediction errors f_{under}.

where again f_{under} is fraction of the jobs that under-predict their waiting time. Thus, these under-predicting jobs wait for fixed resources when they should have run immediately on on-demand resources. As expected, the mean waiting time w increases as f_{under} increases. Since the normalized price under SWW and AJW-T remains the same regardless of the under-prediction error, we omit it here. The graph exhibits the same trend as our model predicts from Figure 6.7 from §6.5.1.1. In addition, our empirical results also emphasize the key point from our model: that it does not alter the normalized price, and it only affects the mean waiting time when the fixed resources are under-provisioned.

LJW Prediction Accuracy. Figure 6.18 plots the normalized price and mean waiting time under LJW policy as a function of the fraction of jobs with inaccurate runtime predictions. For over-predictions, the x-axis represents the fraction of jobs with runtime less than the long job threshold t where we over-predict the running time to be greater than t, while for under-predictions, the x-axis represents the fraction of jobs with runtime greater than t where we under-predict the running time to be less than t. The dotted line shows the price (a) and mean waiting time (b) from LJW with perfect predictions of job running time. As above, our empirical results match



(a) Normalized Price (b) Mean Job Waiting Time

Figure 6.18: Normalized price (a) and mean job waiting time (b) as a function of the fraction of jobs with over- and under-prediction errors (%) in job running time for s=200 `m5.16xlarge VMs` and t=3 minutes when executing our real production batch workload under LJW.

the trends shown by our analytical models in Figures 6.9(a) and 6.9(b) from §6.5.2.1. In particular, our results show that increasing under-prediction errors has little-to-no effect on the normalized price, but results in a linear increase in waiting time. In contrast, increasing over-prediction errors result in a linear increase in price, but a super-linear decrease in mean waiting time.

6.8 Related Works

This work is related to prior work in many different areas.

Cloud Computing. Prior work often assumes the workload is continuous and uniform, rather than composed of discrete jobs, which leads to solutions based on dynamic and integer programming [35, 46, 48, 67, 83, 84]. Thus, this work does not apply to cloud-enabled job schedulers. Our work is also related to prior work on job scheduling for hybrid clouds that run jobs on fixed resources but can also burst into the cloud [41, 57]. While hybrid cloud provisioning and scheduling is well-studied, we know of no work that focuses on explicit waiting policies. As a result, the past works does not explore the key tradeoff between price and job waiting time in hybrid cloud provisioning and scheduling.

Queuing Theory and Marginal Analysis. Our work applies a number of previously developed queuing theory results to gain insights into key tradeoffs exposed by different waiting policies. In particular, our work builds on classic marginal analysis and queuing results by Erlang and others [38, 51, 75, 86], as well prior results on reneging and balking [55]. Many of these models are probabilistic and assume an increasing fraction of customers (or jobs) abandon the queue as their waiting time increases based on diverse customer preferences. These customer-centric models do not apply to our context, where the waiting policy determines whether jobs abandon the queue (and run on on-demand resources).

Ski Rental Problems. Our problem is similar to the classic ski rental problem in online algorithms [14]. However, these assume there is no knowledge of the future, whereas our queueing analysis leverages a workload characterization. Ski rental problems also typically focus on whether to buy or rent a single resource whereas our problem focuses on provisioning, i.e., how many resources to buy versus rent, and generally do not consider the cost and waiting time tradeoff.

6.9 Conclusion and Status

In this chapter, we introduced the concept of a waiting policy for cloud-enabled schedulers, and defines, models, analyzes, and empirically validates multiple fundamental waiting policies. A key goal of this work is to provide a formal foundation for future work on waiting policies both analytically and empirically, including on more general distributions of job inter-arrival and service times, different scheduling policies, and machine learning (ML) classifiers to accurately estimate job waiting and running times. Specifically, our evaluation shows that real workload characteristics differ from our model's assumptions, which motivates analytical models based on more general distributions of inter-arrival and service times.

Status. Waiting policies have been evaluated on a year-long trace from the MGHPCC data center via simulation. Additional details on its design, implementation, and evaluation are in [18, 19].

CHAPTER 7

DATA-DRIVEN JOB SCHEDULING FOR CLOUD-ENABLED SCHEDULERS

Cloud-enabled schedulers execute jobs on either fixed resources or those acquired on demand from cloud platforms. Thus, these schedulers must define not only a scheduling policy, which selects which jobs run when fixed resources become available, but also a *waiting policy*, which selects which jobs wait for fixed resources when they are not available, rather than run on on-demand resources. As with scheduling policies, optimizing waiting policies requires *a priori* knowledge of job runtime. Unfortunately, prior work has shown that accurately predicting job runtime is challenging. In this chapter, we show that optimizing job waiting in the cloud is possible without accurate job runtime predictions. To do so, we i) speculatively execute jobs on on-demand resources for a small time and cost to learn more about job runtime, and ii) develop a ML model to predict wait time from cluster state, which is more accurate and has less overhead than prior approaches that use job runtime predictions.

7.1 Motivation

Batch job schedulers, such as Slurm [8] and LSF [12], execute a large fraction of the workload in high-performance computing (HPC) and data centers. While these schedulers were originally designed to manage a fixed set of servers, they are now generally "cloud-enabled" and capable of autoscaling by programmatically acquiring virtual machines (VMs) from cloud platforms on demand to execute jobs [7]. Thus, these schedulers must not only schedule jobs on fixed resources, but also decide when

to acquire and release on-demand cloud resources. Hybrid clouds often use cloud-enabled schedulers to "burst" into the cloud when their fixed private resources are fully utilized [41].

Cloud-enabled scheduling differs from conventional scheduling on fixed resources in that cost, in addition to job waiting time, is a critical metric. As a result, cloud-enabled schedulers must not only define a scheduling policy, which selects which jobs run when fixed resources become available, but also a *waiting policy*, which selects which jobs wait for fixed resources, and for how long, when they are not available before running on on-demand resources. Waiting policies often mirror traditional scheduling policies, such as Shortest Job First (SJF). Prior work analytically models simple waiting policies, including All Jobs Wait (AJW), No Jobs Wait (NJW), Long Jobs Wait (LJW), and Short Waits Wait (SWW), and shows that combining LJW and SWW offers a much better cost-waiting time tradeoff than the others [18].

Importantly, as with many scheduling policies, optimizing the waiting policies above requires *a priori* knowledge of job runtimes. In particular, LJW directly requires job runtimes, since it forces jobs with runtimes larger than some threshold to wait for fixed resources, but runs shorter jobs immediately by acquiring on-demand resources. Likewise, SWW indirectly requires job runtimes, since it forces arriving jobs expected to wait less than some threshold for fixed resources to actually wait, but runs jobs expected to wait longer immediately by acquiring on-demand resources. Unfortunately, scheduling policies that require knowing job runtimes, such as SJF, are often not widely used because accurately predicting job runtimes remains challenging. Recent work highlights many reasons for the low prediction accuracy, including a lack of sufficient features for training machine learning (ML) models and non-stationarity in workloads that leads to inconsistent performance [53]. Directly implementing the waiting policies above suffers from the same challenges.

The primary contribution in this chapter is showing that optimizing waiting policies for cloud-enabled schedulers is possible without accurate job runtime predictions, and can come close to the cost and waiting time achievable given perfect knowledge of job runtimes. To do so, we develop two techniques to optimize job waiting under LJW and SWW, respectively. Intuitively, optimizing these waiting policies in the cloud is simpler than optimizing scheduling policies for fixed resource because i) there is no hard resource constraint, and ii) our waiting policy predictions require only binary classification, i.e., where a job's running or waiting time crosses a threshold, which does not require absolute model accuracy.

- **Speculative Execution**. We first leverage the availability of on-demand cloud resources to speculatively execute *all* jobs for some time to learn more about each job's running time before deciding whether to run it on fixed or on-demand resources. This technique informs LJW, and is effective because, in many batch workloads, most jobs are short, but the few long jobs account for most of the computation. Thus, the additional cost of speculatively executing the few long jobs incorrectly before restarting them on fixed resources is small.
- **ML-based Waiting Time Predictions**. We next develop a ML model for predicting job waiting time, which can inform SWW. Unlike prior work that uses job runtime predictions to estimate waiting time, e.g., by simulating the schedule based on the runtime predictions [28, 72], our ML model uses cluster state as its input, e.g., cluster size, number of jobs running and waiting, how long jobs have already run and waited, etc. We show that this approach i) has lower computational overhead, ii) is much more accurate, mostly due to the law of large numbers, and iii) is effective at achieving near the cost and waiting time of SWW with perfect knowledge of waiting time.

Our hypothesis is that combining speculative execution and ML-based waiting time predictions can achieve cost and waiting times that are close to optimal LJW and SWW with perfect job running and waiting time predictions.

7.2 Background: Context and Baselines

In this chapter, our methodology is empirical, and focuses on optimizing the workload of a large production batch cluster (same workload trace as in previous chapter§6.6) that services roughly 14 million jobs per year. The cluster currently consists of ∼14.3k cores, uses the LSF job scheduler, and is not cloud-enabled. Our job trace include each job's submission time, user ID, maximum running time limit, requested number of cores and memory, completion status, and running time. While the traces do not record job waiting time, we estimate the average waiting time would be ∼8.1 hours using work-conserving FCFS scheduling, which schedules the first job near the front of the queue capable of running on the available resource. The waiting time under non-preemptive, work-conserving SJF would be much lower at ∼0.6 hours given the workload's large number of short jobs, but requires accurate predictions of job running time.

We consider executing the workload above with a cloud-enabled scheduler on EC2 using the SWW§6.5.1 and LJW§6.5.2 waiting policies combined with either non-preemptive, work-conserving FCFS or SJF scheduling. For fixed resources, we assume the use of 3-year reserved `m5.16xlarge` VMs, which each have 64 cores and 256GB memory. We choose larger VMs to mitigate the impact of imperfect packing of variable-sized jobs onto VMs. Our scheduler packs jobs onto fixed resources using a best-fit policy based on cores. When acquiring on-demand VMs to execute a job, our scheduler selects the smallest and cheapest VM within the `m5` family that satisfies the job's resource requirements.

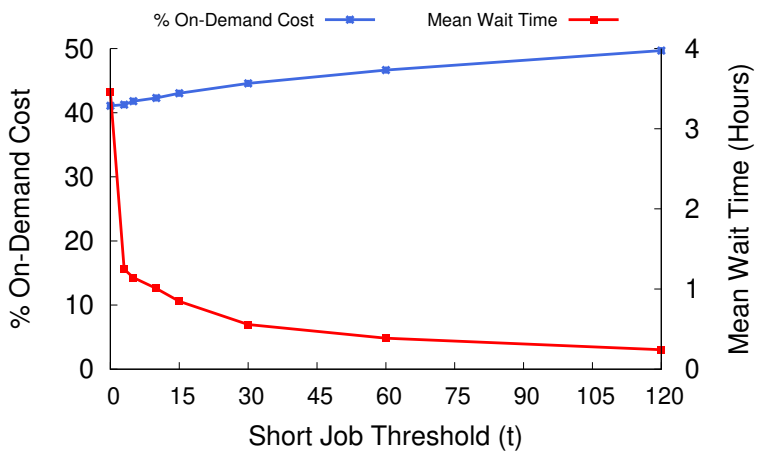


Figure 7.1: On-demand cost, as a percentage of fixed resource cost, (left y-axis) and average waiting time (right y-axis) as a function of LJW's short job threshold t. As t increases, waiting time drops steeply, while cost increases modestly.

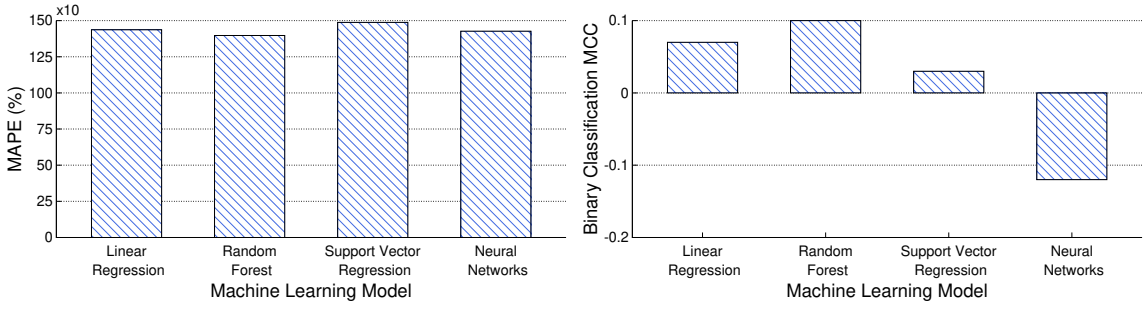
For our baseline, we choose SWW's waiting time threshold b to be 24 hours, and LJW's runtime threshold t to be 15 minutes. Our choice for b is subjective: a higher b decreases cost, but increases waiting time, and there is no optimal value. We chose a 24-hour maximum waiting time because it seems reasonable that no job should wait longer than 1 day to run. Our choice for t=15m is based on Figure 7.1. The graph shows that LJW's cost is mostly flat, while the average waiting time initially decreases sharply and then flattens out. After the initial decrease, LJW's cost-waiting time tradeoff remains relatively constant.

Assuming the waiting policies above with perfect knowledge of job running and waiting times, we empirically determined that the optimal number of `m5.16xlarge` reserved VMs that minimizes cost for our workload was 150 for both SJF (with perfect knowledge of job running times) and FCFS scheduling. That is, adding another reserved VM, as a fixed resource, would not be utilized more than 40% of the time, and would not justify its cost. Thus, we set our baseline for fixed resources s=150. For context, a cloud-enabled scheduler using LJW and SWW parameterized above with s=150 would cost 5% less overall, when combining the amortized fixed resource and on-demand cost, than the current fixed size cluster, which is equivalent to using

225 `m5.16xlarge` VMs, and yield an average waiting time of 0.85 hours when using work-conserving, non-preemptive FCFS scheduling. Finally, while we choose b=24h, t=15m, and s=150 as baselines, our general insights are applicable at any values of these parameters, and especially for smaller s, since, as with scheduling, optimizing job waiting become more important under constraint. Our evaluation varies b, t, and s from our baselines.

7.3 Design

Directly implementing SJF, LJW, and SWW in practice requires predictions of job runtime. To better understand their performance, we first trained and evaluated multiple simple ML models, including linear regression, random forest, support vector regression (SVR), and a neural net, to predict each job's runtime from its characteristics known at submission, as represented in our LSF batch traces. Our models' input features included each job's submission time, user ID, maximum running time limit, and requested number of cores and memory, while the output was the job's running time. We trained the models on data from 10 million jobs over 9 months, and evaluated them on a separate timeframe of 4 million jobs over 3 months.



(a) (b)

Figure 7.2: MAPE (a) and MCC (b) of multiple ML models for predicting job runtime from features in our batch trace.

Figure 7.2(a) shows the results for each model, where the y-axis is the mean absolute percentage error (MAPE) in predicting a job's runtime at submission time. As expected, the error is quite high for all models largely because our batch traces record only a few features for each job, so the models have little data with which to distinguish jobs. Of course, our LJW waiting policy only requires classifying jobs to be above or below some threshold t. Thus, Figure 7.2(b) evaluates these models' binary classification accuracy using the Matthews Correlation Coefficient (MCC), which is the best single measure of binary classification performance. The MCC's values are in the range -1.0 to 1.0, with 1.0 being perfect prediction, 0.0 being random prediction, and -1.0 indicating the prediction is always wrong. The results show that the models are not much better than random predictions, as the MCCs are all near 0.

To get a sense of how effective (or ineffective) such models are in practice, we used the best model above to simulate SJF on our current fixed-size cluster. Recall from §7.2 that the average job waiting time under SJF with perfect knowledge of job runtime is 0.6 hours on the current fixed-size cluster (equivalent to 225 `m5.16xlarge` VMs). However, simulating SJF using our ML model to predict job runtimes results in an average waiting time nearly 3× higher at 1.71 hours. For context, a random job next policy yields an average waiting time of 3.1 hours, so our job runtime prediction model yields an average waiting time for SJF roughly mid-way between using perfect predictions and random predictions. Interestingly, despite our ML model's poor prediction accuracy, it does appear to have better accuracy with respect to ordering jobs. Note that a random job next policy has a much lower waiting time than FCFS (at ∼8.1 hours) because it is more likely to select one of the large number of short jobs to run.

While the ML models above are simple, and may not be the most accurate, they illustrate long-standing issues with predicting job runtime in batch workloads. As we discuss, improving the accuracy of these models is *not necessary* to optimize job

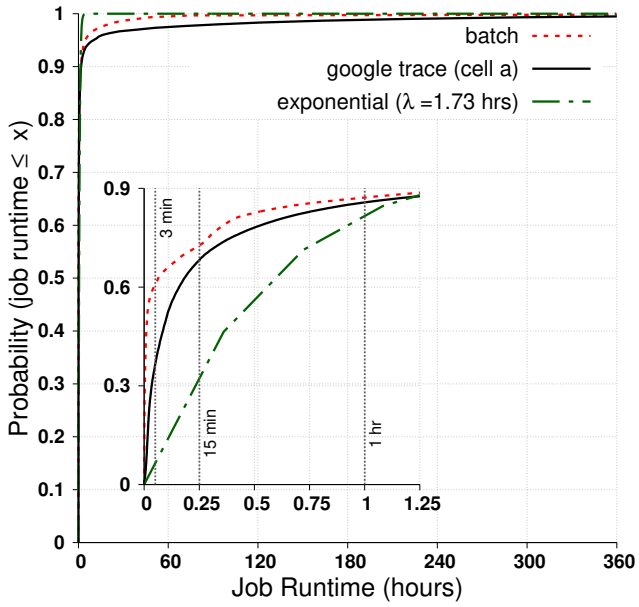


Figure 7.3: CDF of job runtime for our batch workload, a widely-used Google job trace [76], and an exponential distribution with the same mean as our batch workload.

waiting for cloud-enabled schedulers. This is due, in part, to LJW and SWW's use of a threshold to make decisions. LJW classifies a job as long, if its running time exceeds a threshold t, and as short otherwise, while SWW similarly classifies a job's waiting time as short if it is less than b. In both cases, the decision depends on whether the estimated value is above or below a threshold. Thus, even when job runtimes cannot be estimated (or even ordered) accurately, the waiting policy is correct as long as the job (in LJW) or wait time (in SWW) is estimated accurately with respect to its threshold.

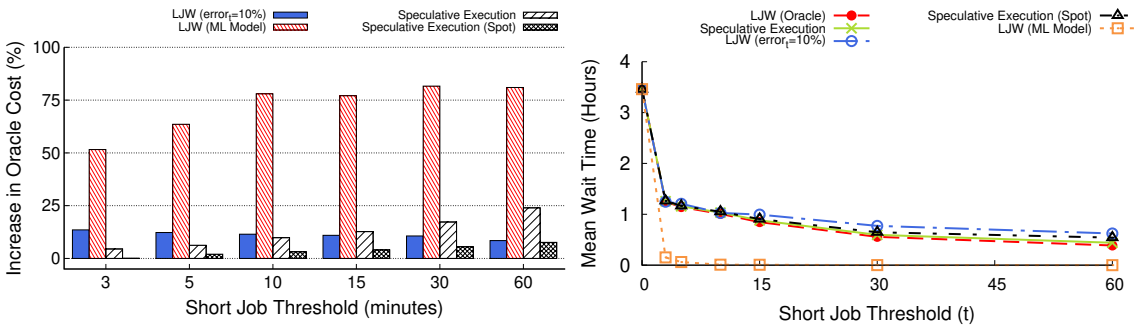
7.3.1 Optimizing LJW using Speculative Execution

Cloud-enabled schedulers can always acquire on-demand resources at an additional cost to execute jobs. We leverage this capability to optimize the LJW waiting policy for a small additional cost. As with SJF, LJW requires *a priori* knowledge of job running time to make decisions about which jobs wait, and which jobs run, on on-demand resources. Instead of using a job runtime prediction model like those above

to make this decision, our approach is to acquire on-demand resources to run *all* jobs immediately if no fixed resources are available when a job arrives. If a job's running time is less than t, then it will simply complete without incurring any additional cost or waiting time compared to if the scheduler had perfectly predicted its runtime. However, once a job runs on on-demand resources for t time, based on LJW, it is classified as a long job and the scheduler kills the job, releases the on-demand resources, and queues the job to run on fixed resources. We do not assume the scheduler can checkpoint, migrate, and restore jobs, since most schedulers do not support it.

The benefit of speculative execution for LJW is that it always handles short jobs correctly, by running them on on-demand resources to completion (when fixed resources are unavailable), but it incurs an additional cost of $t \times p$ for long jobs compared to if the scheduler had perfectly predicted job runtimes. Cloud-enabled schedulers can effectively leverage speculative execution to buy some information about job running time. The approach is cost-effective in practice if most of the jobs are short, but most of the resources are used by long jobs. This is case for many batch workloads, including ours. Figure 7.3 shows a cumulative distribution function (CDF) of job runtimes for our cluster's workload and a widely-used publicly-available Google trace [76] with dotted vertical lines at runtimes of 3 minutes, 15 minutes, and 1 hour. The Google trace exhibits the *same* characteristics and general trend as our batch workload. While this skewed runtime distribution makes it challenging to identify the few long jobs, it also makes speculative execution cost-effective. Since both real-world workloads above have a significantly higher fraction of short jobs than the exponential distribution, speculative execution in practice is likely to be much more effective than queuing models might suggest [18].

To illustrate, Figure 7.4 (a) plots the LJW threshold t on the x-axis, and the increase in on-demand VM cost, as a percentage of the same cost when using LJW



(a) (b)

Figure 7.4: Percentage increase in cost compared to using LJW with a job runtime prediction oracle (a) and average waiting time (b) for multiple LJW variants and and multiple short job thresholds (x-axis).

with an oracle that has perfect knowledge of job runtime predictions. This graph uses non-preemptible, work-conserving FCFS scheduling. The graph compares the oracle with four different approaches: LJW using our best ML model from Figure 7.2, LJW using a hypothetical ML model with 10% average error in job runtime predictions, speculative execution on on-demand VMs, and speculative execution on spot VMs. The graph shows that LJW using our ML model performs the worst, resulting in a 50-75% increase in the cost of using on-demand VMs. In contrast, speculative execution on on-demand VMs incurs a much lower cost, especially for small thresholds. Speculative execution has a similar cost to using a hypothetical job runtime prediction model with 10% average error at our baseline of t=15 minutes (and below). We can further decrease the cost of speculative execution using spot VMs instead of on-demand VMs (we use the spot revocation model as [52]). As t increases, speculative execution's cost also increases, as there are fewer jobs at these longer durations.

Figure 7.4 (b) similarly plots the corresponding average waiting time for each scenario. The graph shows that the average waiting time decreases as the LJW threshold t increases in all cases. LJW using our ML model has a near zero waiting time because it predicts nearly all jobs are short, and thus always runs them on on-demand VMs without waiting, which incurs a high cost. The other approaches yield

an average waiting time close to, but slightly higher than, LJW using an oracle. The results above indicate that, for batch workloads with job runtime distributions skewed towards short jobs, speculative execution mitigates the need to accurately predict job runtimes, and that such predictions would need to have less than 10% error to yield a similar cost and waiting time (for our baseline parameters). Achieving such low model error in practice is unlikely.

7.3.2 Optimizing SWW using Machine Learning

We next focus on optimizing SWW using an ML-based model for predicting a job's waiting time. There is less prior work on predicting waiting times for conventional schedulers, since these predictions typically serve only to inform users, but generally do not improve scheduling for fixed resources. However, prior work has explored predicting queue waiting times within the context of certain scenarios where it is more than just informative, such as when users can choose between multiple queues or clusters [28, 58], or when users can alter their requested resources *post facto* to shorten waiting time [73]. This prior work focuses on conventional scheduling for fixed resources, and not cloud-enabled scheduling.

A common approach for estimating queue waiting time is to use predictions of job running time to simulate the schedule forward [59, 72]. This approach, of course, is dependent on accurate job runtime predictions, which, as we discuss above, are often not available. In addition, for "optimal" scheduling policies, such as SJF, these approaches also depend on future job arrivals that are not represented when simulating a schedule. While, in this case, ML-based and other statistical approaches suffer from the same limitation, waiting policies can mitigate the difference in waiting time between SJF and FCFS scheduling, which can motivate the use of a simpler scheduling policy like FCFS. Below, we focus on estimating wait times using FCFS scheduling, which are only dependent on the jobs currently in the system.

Feature	Description
cluster-cpu-util	Average CPU utilization of fixed resources
cluster-mem-util	Average memory utilization of fixed resources
runq-size	Number of running jobs on fixed resources
waitq-size	Number of jobs waiting for fixed resources
runq-mean-cpu	Mean CPU resource demand of jobs running on fixed resources
runq-mean-time	Mean time running (up to now) for jobs running on fixed resources
waitq-mean-cpu	Mean CPU resource demand of waiting jobs
waitq-mean-time	Mean time waiting (up to now) for jobs in queue
num-cores	Number of CPU cores requested by the new job

Table 7.1: Cluster state features used for training our ML-based waiting time prediction models

Rather than rely on job runtime predictions, we instead train a ML model to predict job waiting time based on cluster state, including the number of queued and running jobs, average size of queued and running jobs, average time of running jobs, etc. Our intuition is simple and derives from the law of large numbers: an ML model for predicting job wait time should be more accurate than for predicting job runtime, especially for large clusters, as the former depends on the average runtime of a large number of jobs, while the latter depends on a single job. While any single job's runtime represents just one sample from a workload's job runtime distribution, a large number of queued jobs represents a much larger sample and thus their average job runtime is more likely to be closer to the mean of the job runtime distribution, which determines, in part, a job's wait time.

We use the intuition above to train three different ML models using the features in Table 7.1, namely linear regression, random forest, and gradient boosting. We simulate execution of the batch trace on our cluster under SWW using our baseline parameters, i.e., b=24h, s=150 `m5.16xlarge` VMs, with work-conserving, non-

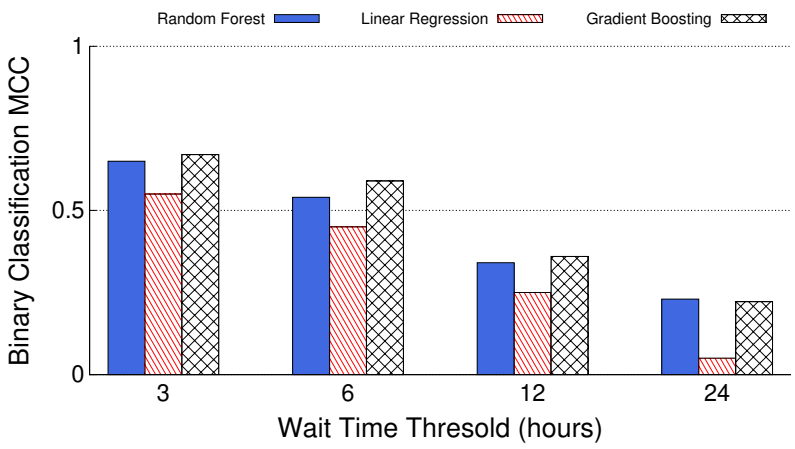


Figure 7.5: MCC of different ML models for predicting job waiting time for different waiting time thresholds b.

preemptive FCFS scheduling. For each job, we then record the features from Table 7.1 at submission time, and then record its waiting time once it is scheduled. Note that *runq-mean-time* and *waitq-mean-time* are jobs' running and waiting times, respectively, up to the present, and so jobs' final running and waiting time may be longer. While our problem is a binary classification, i.e., is the waiting time longer than b, we train multiple regression models using these features to avoid re-training for new values of b.

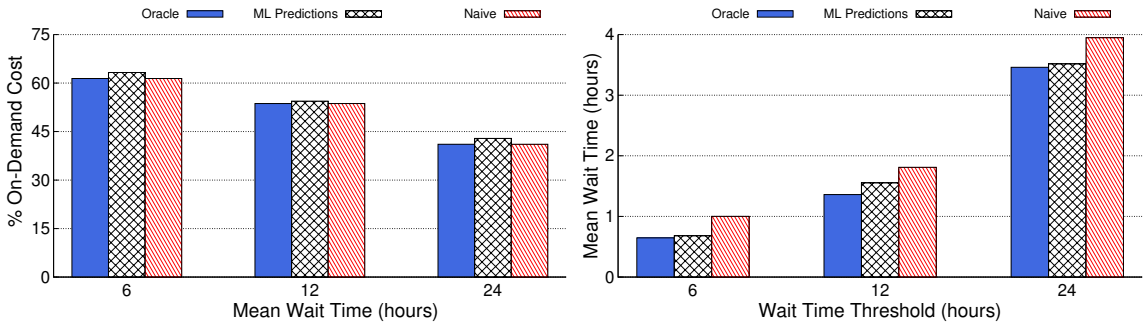


Figure 7.6: On-demand cost, as a percentage of fixed resource cost (a) and average job waiting time (b) for different approaches to predicting job waiting time under SWW with different thresholds b (x-axis).

Figure 7.5 plots the MCC of our binary classification on waiting time for our baseline b=24h under the different regression models for different values of b. The graph shows that random forest and gradiant boosting have MCCs 0.25-0.7 with the MCCs increasing as the threshold b decreases. By contrast, a naïve approach that forces all jobs to wait yields an MCC of 0. Note that an approach that estimates job waiting time by simulating the schedule forward using our job runtime prediction model from §3.2 behaves similarly to such a naïve all-jobs-wait approach, since it tends to under-predict each job's waiting time.

We integrated the random forest model above into our simulator to predict job waiting times, and compared its performance both to using an oracle with perfect knowledge of job waiting times, and to the naïve approach above. Figure 7.6 (a) shows the results for different values of the SWW threshold b along the x-axis, and the additional on-demand cost, as a percentage of the cost of fixed resources, on the y-axis. Here, we again use s=150 `m5.16xlarge` VMs as the number of fixed resources. The graph shows that our ML-based model yields a cost within 2% of the oracle at our baseline b=24h. Note that the naïve approach yields the same cost as the oracle, by definition, but has a mean waiting time that is 14% higher than the oracle at our baseline b=24h, as shown in Figure 7.6 (b). In contrast, our ML-based waiting time predictions have a waiting time much closer to oracle, >1% at our 24h baseline, and essentially equal at b=6h.

7.4 Implementation

We extend the trace-driven cloud-enabled job scheduling simulator from §6.6 to implement speculative execution and integration of ML models in waiting policies. The simulator supports either work-conserving SJF or FCFS scheduling, and the LJW and SWW waiting policies. For SJF, LJW, and SWW policies, the simulator uses an API to fetch job runtime and waiting time from a model. We can specify whether

this model is an oracle, or one of the ML-based models we describe in the previous section. We can also specify the short job threshold t (for LJW) and waiting time threshold b (for SWW) at startup. The simulator tracks statistics including average job waiting time, on-demand cost, and average fixed resource utilization.

Our evaluation focuses on two large-scale traces that are representative of job scheduling in academia and industry. Our academic job trace, which we describe in §6.6, is from a shared cluster from a large university system that covers multiple campuses, and thus encompasses the full spectrum of jobs submitted by the medical, science, and engineering research communities. Our industry trace is an updated release of the widely-used and publicly-available Google cluster trace, and is an order of magnitude larger [76]. Google uses the Borg scheduler, and we use a portion of the trace that includes 58 million jobs over one week run on a single Borg cell. Note that the Borg scheduler manages both batch and service jobs, where the latter are resource requests for interactive services which typically cannot be arbitrarily delayed [81]. However, since the Google trace does not specify the type of job, we treat them all as delay-tolerant.

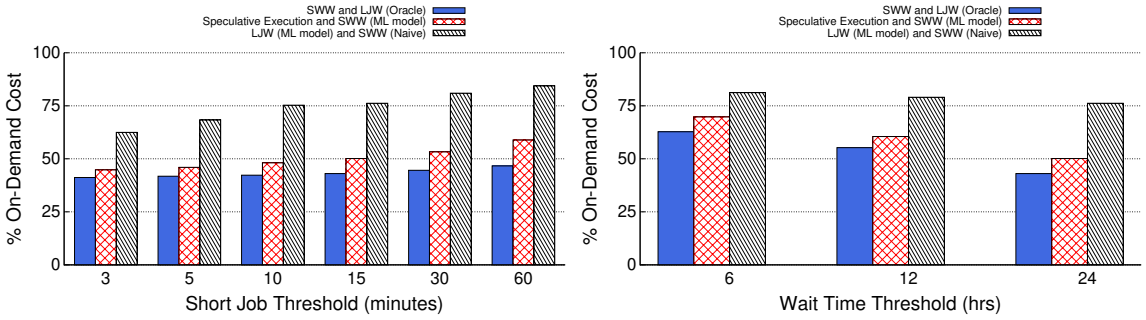


Figure 7.7: On-demand cost, as a percentage of fixed resource cost, on the y-axis as a function of both LJW's short job threshold t (a) and SWW's waiting time threshold b (b) using our baseline parameters.

We reference a number of ML models in both the previous and next section, which we have trained using the traces in conjunction with our simulator. We use python's

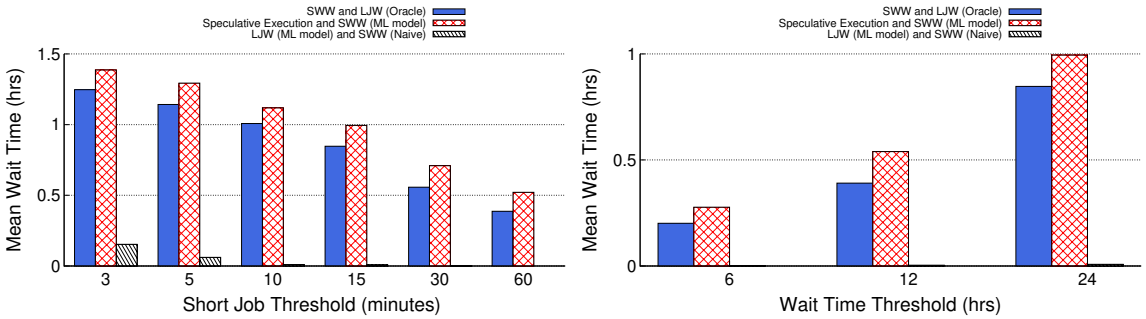


Figure 7.8: Mean wait time (hours) on the y-axis as a function of both LJW's short job threshold t (a) and SWW's waiting time threshold b (b) using our baseline parameters.

scikit-learn [29] module for training, and focus on basic models. Our job runtime prediction models are directly trained from the features known at submission time in the trace data above. For our waiting time predictions, we use our simulator to generate a new trace that records the cluster state from Table 7.1 at each job submission, and then records the job's waiting time once it is scheduled. Of course, this waiting time in the generated dataset is dependent on the number of fixed resources s we configure for our simulator. To generate this new dataset, we use a work-conserving, non-preemptive FCFS scheduling policy with LJW and SWW using our baseline parameters from §7.2. We use this as training data to learn our models of job waiting time. For training our job waiting time ML model, we use 70% of the dataset and 30% of the dataset for testing the models. In addition, we use simple hyperparameters for tuning our ML models, specifically a tree depth of 114 and a random seed of 137 for both random forest and gradient boosting trees.

7.4.1 Real Implementation

We describe an implementation of a prototype batch computing service that implements various waiting policies. Our service would be implemented as a lightweight and extensible wrapper to the Slurm workload manager. Interestingly, Slurm already supports basic waiting policies like no jobs wait (NJW) and all jobs wait (AJW).

For example, NJW policy is similar to existing auto-scaling policies for cloud-enabled schedulers, and Slurm already provides functions to support such auto-scaling policies. On the other hand, Slurm doesn't directly support waiting policies like all short waits wait (SWW) and long jobs wait (LJW) as they require predicting job runtimes and wait times.

To implement the waiting policies in our prototype, we would utilize several of Slurm's in-built features like monitoring cluster state, monitoring job status, cluster autoscaling, partitions, etc. In particular, we would define multiple partitions to manage the cluster (where partitions can be considered as job queues, each of which has an assortment of constraints such as job size limit, job time limit, users permitted to use it, etc.). To start, we would have two partitions (with no explicit runtime limit) – fixed partition (which runs jobs on fixed or reserved resources) and on-demand partition (which executes jobs on on-demand VMs and autoscaling is enabled). Note that our service would monitor cluster state, job completions, and failures through the use of Slurm call-backs.

Speculative Execution. For implementing speculative execution, we would configure the cluster to use three partitions. In addition to the above two partitions, speculative execution requires a new partition that executes jobs on on-demand VMs immediately with an explicit runtime limit (set to long job threshold). When all fixed resources are exhausted, then new jobs will launch using this new partition. Our prototype monitors the terminated jobs due to time limits and then re-queues or relaunches the jobs using fixed partition.

SWW using ML-based wait time predicitions. To implement SWW using wait time predictions, our prototype would monitor system state using Slurm call-backs and uses this information to predict each job's waiting time on arrival. If the predicted wait time is lower than the wait time threshold, then we execute the job on the fixed

resources eventually. If not, we launch the job immediately on on-demand VMs (using the on-demand partition).

7.5 Evaluation

Our evaluation focuses on i) combining the techniques from §3.2 to quantify how close the cost and waiting time come in practice to that of an oracle; ii) quantifying the effect of the number of fixed resources s on the magnitude of the results; and iii) showing that these techniques also generalize to the Google trace, which has similar job runtime characteristics.

7.5.1 Combining Techniques

Figure 7.7 shows the on-demand cost, as a percentage of fixed resource cost, on the y-axis as a function of both LJW's short job threshold t (a) and SWW's waiting time threshold b (b) using our baseline parameters. We compare three techniques: SWW and LJW using an oracle with perfect knowledge of job waiting and running time; a naïve approach that uses an ML model for predicting job runtimes for LJW and SWW; and our techniques from §3.2 that use speculative execution and an ML-based waiting time prediction model. In both graphs, our techniques come much closer to the cost of the oracle compared to those using predictions of job runtime across all short job thresholds t and waiting time thresholds b. Specifically, at our baseline parameters of (t=15m, b=24h) the combined technique comes within 4% of the oracle's on-demand cost. By comparison, using job runtime predictions has a 70% higher cost compared to the oracle. The cost advantage is similar across all parameter settings.

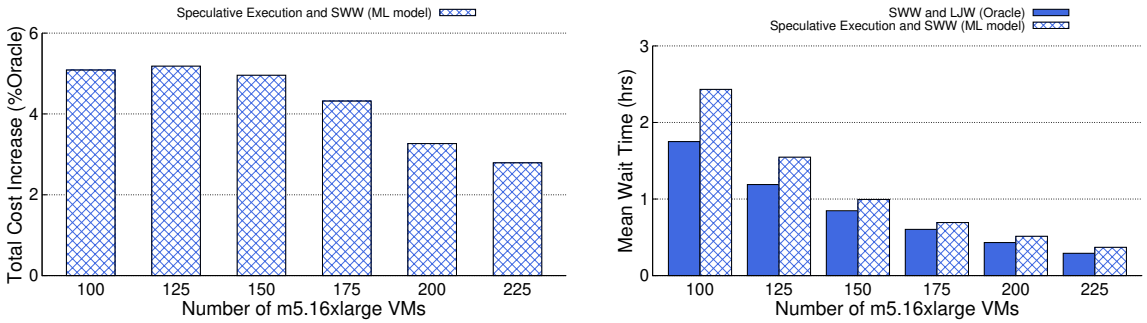
Figure 7.8 similarly shows the mean wait time on the y-axis as a function of both LJW's short job threshold t (a) and SWW's waiting time threshold b (b) using our baseline parameters. This is the same experiment as in Figure 7.7. Again,

combining our techniques from §3.2 of speculative execution and ML-based waiting time predictions results in a waiting time near that of the oracle across all short job thresholds t and waiting time thresholds b. For our baseline parameters (t=15m, b=24h) combining our techniques comes within 13% of the oracle's mean waiting time. In contrast, a policy that directly uses job runtime predictions for LJW and SWW has nearly zero waiting time because it tends to under-predict job running time due to the large number of short jobs. As a result, it runs most jobs on on-demand resources at a high cost, but with low waiting time.

Recall from §7.2, that the waiting time on the current fixed-size cluster (equivalent to 225 `m5.16xlarge`'s using SJF with perfect knowledge of job running time is 0.6 hours, but is 1.71 hours in practice, when using our job runtime prediction model from §3.2. For this experiment, the average waiting time across all the parameters are less than 1.71 hours, and many are less than 0.6 hours. Of course, the maximum waiting time in our case is bounded by the waiting time threshold b, while the maximum waiting time is unbounded under SJF. Recall also that the total cost, including both fixed and on-demand resources, of using LJW and SWW under an oracle with our baseline parameters is 5% less than the cost of current fix-sized cluster, and our practical approach achieves near this cost. This shows how optimizing waiting policies for cloud-enabled schedulers can mitigate some of the challenges with optimizing scheduling policies.

7.5.2 Varying Fixed Resources

Up to this point, all of our experiments have used the same number of fixed resources s of 150 `m5.16xlarge` VMs, which is optimal number of fixed resources for our workload that minimizes the total cost of fixed and on-demand resources, when amortized over the workload's year-long duration. In this case, we assume the cost of fixed resources is equivalent to the price of 3-year reserved `m5.16xlarge` VM.



(a) Cost (b) Wait time

Figure 7.9: Total cost of amortized fixed and on-demand resources (as a percentage of the oracle) as a function of fixed resource capacity (a). Mean wait time as a function of fixed resource capacity for our approach and the oracle (b).

Figure 7.9 shows the impact of varying the number of fixed resources on both the cost and waiting time, where are other baseline parameters remain the same. In this case, Figure 7.9(a) includes the total fixed and on-demand cost for executing the workload under SWW and LJW, as a percentage of the oracle. The graph shows that speculative execution and ML-based waiting time predictions achieves near the same total cost, regardless of number of fixed resources. Note that our cost is closer to the oracle at 150 VMs than above because the previous section only plotted the on-demand cost assuming that fixed resources were a sunk cost.

Figure 7.9(a) shows that as we increase the number of fixed resources, the average waiting time decreases, as expected, although the percentage difference between our approach and the oracle increases. However, ultimately, the importance of waiting policies decreases as fixed resources increase, since there is less resource constraint and need to wait.

7.5.3 Generalizing to the Google Workload

Our illustrative examples in §7.3 and evaluation above are from a single workload. To demonstrate the generality of our approach, we performed a similar evaluation using the Google trace [76]. The trace includes data from 8 Borg cells over a single

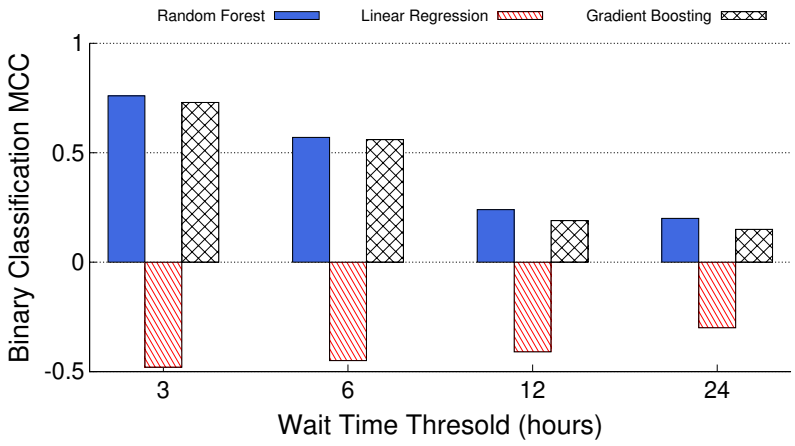


Figure 7.10: MCC of ML models for predicting job waiting time for different waiting time thresholds b in the Google trace.

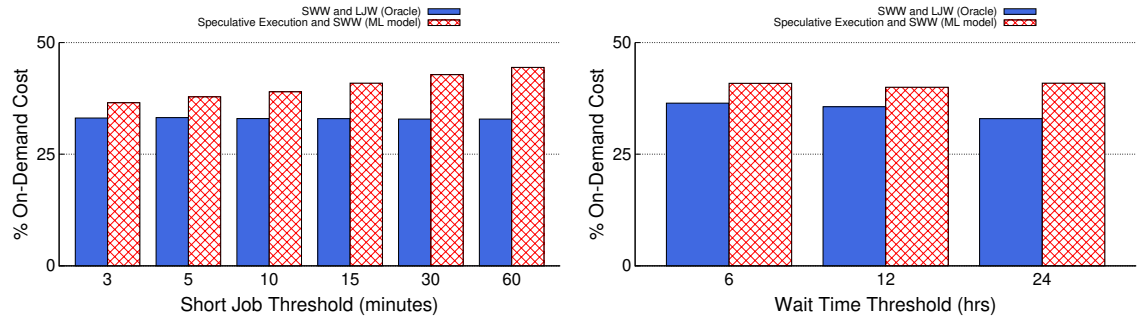


Figure 7.11: On-demand cost, as a percentage of fixed resource cost, on the y-axis as a function of both LJW's short job threshold t (a) and SWW's waiting time threshold b (b) for our Google trace using the baseline parameters.

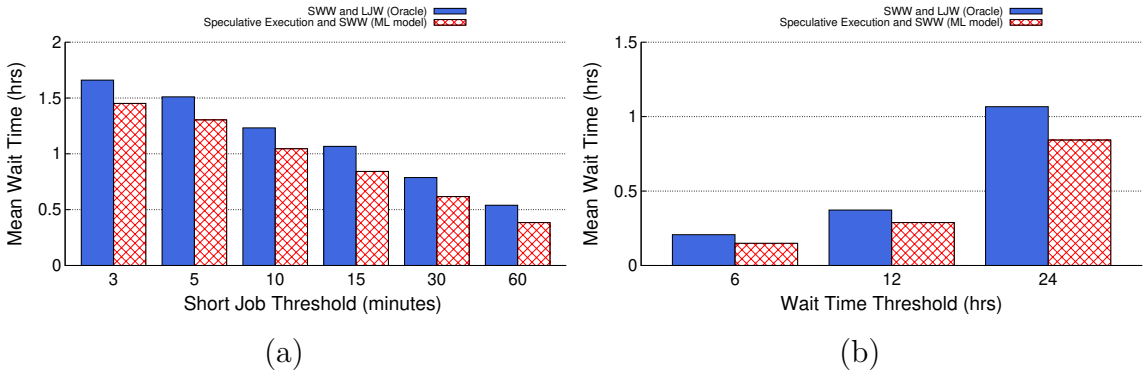


Figure 7.12: Mean wait time (hours) on the y-axis as a function of both LJW's short job threshold t (a) and SWW's waiting time threshold b (b) for Google trace using our baseline parameters.

month in May 2019. Since the number of jobs is massive, we focus on a single week from a single cell, which includes 58 million job submissions. We further randomly sample this down to 14 million job submissions, or 25%, to reduce the overhead of our simulations. Note that our sampled trace has the same mean core/memory request and job runtime as the original trace. The jobs in the Google trace have much larger core/memory requirements than our academic trace, so we adjust the number and size of our baseline fixed resources for this evaluation. We set our baseline number of fixed resources s equal to 4000 VMs, each with 192 cores and 768GB memory.

We use the same per-core pricing as in the previous experiments, which is based on the `m5` family, i.e., $0.048 per core-hour. The `N2` family of VMs in Google Compute Engine (GCE) have a similar price. In this case, for on-demand VMs, we assume the use of custom VMs from GCE with $0.031611 per core-hour and $0.004237 per GB-hour, as the Google trace has many jobs with unbalanced core/memory ratios that waste significant resources when using fixed-size VMs. These custom prices are equivalent to $0.048 per hour for 1 core and 4GB memory, as above. As before, we assume the amortized cost of the fixed resources has a 60% lower cost, equivalent to that of a 3-year reserved VM. We use the same baseline parameters as in the other analysis (t=15m, b=24h).

Recall from Figure 7.3 in §7.3 that the Google workload's job runtime distribution is remarkably similar in shape to that of our academic batch trace, where a significant fraction of jobs are short, but where much of the computation comes from a small fraction of long jobs. We also trained waiting time prediction models using the same approach as in §6.5.1 by generating a dataset from a simulation run that recorded the features listed in Table 7.1. Figure 7.10 plots the Matthews Correlation Coefficient (MCC) for these models. The results for a random forest and gradiant boosting model are similar to those in our academic workload, from Figure 7.5, while linear regression

actually exhibits a negative MCC. We use a random forest model, since it yields the highest MCC.

Figure 7.11 shows the on-demand cost as a percentage of fixed resource cost, on the y-axis as a function of both LJW's short job threshold t (a) and SWW's waiting time threshold b (b) for our Google trace using the baseline parameters. Figure 7.11(a) shows the same trends we observed in Figure 7.7(a) for speculative execution as the short job threshold t changes. For small values of t, the difference in cost between the oracle and speculative execution is minimal because a large fraction of jobs are short, and thus *should* run on on-demand VMs. Similarly, Figure 7.11(b) also shows the same trend observed in Figure 7.7(b) as we change the waiting time threshold b, where slightly shorter thresholds have a cost closer to the oracle. Again, the observation that the accuracy of waiting time predictions based on cluster state is aided by the law of large numbers is general, and holds for the Google trace just as it does for the academic trace.

Figure 7.12(b) shows the average waiting time for the same experiment as above. We see similar trends as in Figure 7.8(b) using the academic workload, except that the waiting time for our approach in the Google trace is actually slightly less than when using an oracle. This occurs because our ML-based waiting time prediction model performs slightly better compared to these models for our academic trace. As a result, there are fewer jobs that actually end up waiting for fixed resources for time b in the queue, and then also incur the high price of using on-demand VMs. These waiting time mis-predictions are the reason our academic batch workload has both a slightly higher cost and waiting time compared to the oracle. When using an oracle, cost and waiting time are a tradeoff: using more on-demand VMs incurs a higher cost, but should lower waiting time, since there is no reason to wait for an on-demand VM with an oracle. However, mis-predictions can cause waiting for on-demand VMs in practice, which can increase average waiting time. This happens

much less in the Google trace compared to the academic batch workload, and thus higher cost compared to the oracle is compensated by a lower average waiting time.

7.6 Related Works

Conventional job scheduling on fixed resources has been studied for decades, and continues to be an active area of research [50, 60, 78]. Prior work has examined the problem in many contexts, e.g., with deadlines, priorities, fairness constraints, etc. As more computation shifts to cloud platforms, conventional job scheduling is becoming less important for cloud users, since clouds provide the illusion of infinite scalability.

Recent work introduced the waiting problem for cloud-enabled schedulers, and analyzed it using a $M/M/s$ queuing model [18]. However, that work focused on optimizing fixed resource provisioning to minimize cost, and showed that the optimal was dependent on the waiting policy. However, the work assumed waiting policies with perfect knowledge of job running and waiting times. We, instead, show that we can realize these waiting policies in practice without perfect knowledge by using speculative execution and ML-based waiting time predictions. Despite its importance to cloud scheduling, we have not seen any other prior work that directly addresses waiting policies. Our work is related to prior work on scheduling for hybrid clouds, which include fixed private resources, but can also burst into the cloud [41, 57]. However, that work does not define the notion of a waiting policy.

The focus of this work is to demonstrate that we can realize waiting policies in practice that are close to the optimal, given *a priori* knowledge of job running and waiting time. There has been substantial prior work on predicting job running and waiting time for cluster job schedulers, although much of it is not used in practice [36,37,40,59,77]. For example, [77] uses a clustering approach that groups jobs by their attributes and then predicts job runtime within each group. Recent work details the many reasons why cluster schedulers do not use job running time predictions [53],

including low accuracy due to insufficient data, non-stationarity, and unfair performance. Our work echoes many of the same points, as standard ML models cannot even accurately categorize job running times to be above or below a threshold.

There is much less work on predicting job waiting time because it does not directly benefit conventional scheduling [28, 72]. Prior work generally builds on job runtime predictions, and we know of no work that directly uses cluster state. None of this work applies these prediction methods to waiting policies, but instead looks at various other scenarios where jobs have a choice among multiple queues or can modify their request to reduce waiting time. Our focus is not necessarily on developing the most accurate waiting time prediction model, but instead to show that basic models can do well in the context of waiting policies largely due to the law of large numbers.

7.7 Conclusion and Status

This chapter focuses on optimizing the tradeoff between cost and waiting time for cloud-enabled schedulers, which can run jobs on both fixed resources and those acquired on-demand from cloud platforms. This tradeoff is dependent on the scheduler's *waiting policy*, and optimizing the waiting policy generally requires *a priori* knowledge of job runtime. We present two techniques—speculative execution and ML-based waiting time predictions—that enable implementing near-optimal waiting policies in practice without accurate job runtime predictions. We evaluate these techniques on two large job traces from academia and industry, and show they yield a cost and waiting time near that of an oracle with perfect knowledge of job running and waiting time.

Status. We have implemented both speculative execution and ML-based waiting time predictions in a trace-driven job scheduling simulator and evaluated using an academic workload trace and an industry workload trace. We have submitted the work to the ACM SoCC conference in May 2021.

CHAPTER 8

CONCLUSION

Cloud platforms offer the same VMs under a variety of purchasing options that specify different costs and time commitments, such as on-demand, reserved, transient, and spot block. While these options provide opportunities for optimizing the long-term cloud costs (and performance), but choosing from among these options can be challenging. In this work, we focus on optimizing the long-term cloud cost by judiciously selecting different VM purchasing options. As part of our work, we identify multiple significant problems and design system-level solutions using classic statistics when applicable and evaluate these solutions in a real-world scenario to illustrate the benefits of proposed solutions.

8.1 Summary of Contributions

First, we presented TR-Kubernetes, a minimal extension of Kubernetes that executes mixed interactive and batch workloads on unreliable transient VMs dynamically acquired from cloud platforms. To achieve this, we design a greedy provisioning algorithm that satisfies a capacity availability requirement at a low cost. We implemented TR-Kubernetes prototype on EC2's variant of transient VMs and evaluated its performance, reliability, cost, and availability using publicly available benchmarking tools, availability and cost data, and workload traces. Evaluation using Amazon EC2 spot data shows that, when compared to running interactive services on on-demand VMs, TR-Kubernetes is capable of lowering costs (by 53%) and providing higher availability (99.999%).

Second, we analyze the speedup and cost of executing parallel batch jobs, such as distributed ML jobs, on highly discounted transient cloud resources using many different straggler mitigation techniques. Using this model, we derive the expected running time and cost for straggler mitigation techniques proposed in prior work for a simple parallel job with synchronization barriers. Our analysis shows that transient VMs offer complex tradeoffs compared to using on-demand VMs, and can result in higher overall costs despite their highly discounted price due to their probabilistic performance.

Third, we design multiple policies to optimize long-term cloud costs by selecting a mix of VM purchasing options based on short- and long-term expectations of workload utilization with no job waiting. We evaluate our policies on a batch job trace spanning 4 years from a large shared cluster for a major state university system that includes 14k cores and 60 million job submissions and show how these jobs could be cost-effectively executed in the cloud using our approach. Our results show that our policies incur a cost within 41% of an optimistic offline optimal approach, are 50% less than solely using on-demand VMs, and 79% less than using reserved VMs.

Finally, we introduce the concept of a waiting policy for cloud-enabled schedulers. In this work, we defined, analyzed, and empirically validated multiple fundamental waiting policies. Our analysis reveals key tradeoffs in designing waiting policies under FCFS scheduling, and also captures the impact of inaccurate predictions of job running time and waiting time on the fixed resource provisioning, price, and mean waiting time. A key goal of this work is to provide a formal foundation for future work on waiting policies both analytically and empirically, including on more general distributions of job inter-arrival and service times, different scheduling policies, and machine learning (ML) classifiers to accurately estimate job waiting and running times. In addition, waiting policies are important in understanding how users value and provision fixed and on-demand resources. Understanding these user valuations

is important for cloud providers in determining how to set the price of fixed and on-demand resources to maximize their revenue.

The optimal waiting policies generally requires *a priori* knowledge of job runtime which is a complex and non-trivial task using the publicly available dataset on job scheduling. To address this, we present two techniques—speculative execution and ML-based waiting time predictions—that enable implementing near-optimal waiting policies in practice without accurate job runtime predictions. We evaluate these techniques on two large job traces from academia and industry and show they yield a cost and waiting time near that of an oracle with perfect knowledge of job running and waiting time.

8.2 Directions for Future Research

Our dissertation focused on selecting different VM purchasing options for optimizing the long-term cloud costs from a *cloud user's* perspective. Here, we identify several research directions to extend our work.

Combining Waiting Policies with Resource Overcommit. In the recent past, multiple papers [20,26] propose different techniques to improve the resource utilization in datacenters from *cloud platform's* perspective using resource harvesting, resource overcommit, etc. In these works, authors address the issue of low resource utilization by either overcommitting the resources i.e., the sum of resources allocated to the tasks on a machine exceeds its physical capacity, or by harvesting idle resources using dynamically sized harvest VMs. Recall, waiting policies yield a nice trade-off between cost and waiting time, whereas overcommit algorithms typically yield high resource utilization and lower operating cost. Certainly, combining overcommit, waiting policies, and even resource harvesting could potentially gain the benefits of all three. The challenge would be if there are any conflicts between the policies.

Another interesting aspect would be quantifying the relative importance of each one in assessing cost and waiting time.

Energy Efficient Resource Provisioning. Given the importance of building green and energy-efficient data centers for reducing the carbon impact, a natural direction for future work would be to incorporate *energy effecieny* as another constraint while provisioning resources in several of our works. We can directly extend our work on waiting policies, where we can design a carbon-aware scheduler that implements the waiting policies. In this case, using renewable energy is the equivalent of using a cheap fixed resource (which represents a sunk cost), and using grid energy is the equivalent of using expensive on-demand servers. Certainly, having jobs wait for renewable energy is going to lower their carbon footprint. Of course, a key difference is that, in this case, our cheap fixed resource has a variable capacity, and we replace cost with carbon. If the carbon footprint of grid energy is also variable, then it might be beneficial, in terms of carbon, to wait some for it as well.

Optimal VM Pricing from Cloud Provider's Perspective In general, cloud providers set VM prices based on various factors like hardware cost, operation cost, provider benefit, uncertainty in demand, etc. (e.g., reserved VMs are cheaper because they have a better guarantee). An interesting problem about VM pricing from a cloud provider's perspective is – *how to set VM prices so that they can optimize (or maximize) their benefit assuming that rational users act optimally under a given waiting policy?*. To address the problem, we can model and solve with these variables given – hardware cost, operational cost, waiting policy and its thresholds, number of users, etc. The challenge here would be accounting for the hardware and operational costs accurately as it is much harder for us to model from the outside.

BIBLIOGRAPHY

[1] Azure Kubernetes Service. https://azure.microsoft.com/en-us/services/container-service/, Accessed August 2017.

[2] Amazon Elastic Container Service for Kubernetes. https://aws.amazon.com/eks/, Accessed May 2018.

[3] Docker Swarm. `https://docs.docker.com/engine/swarm/`, Accessed April 2018.

[4] Kubernetes on AWS. https://kubernetes-incubator.github.io/kube-aws/, Accessed May 2018.

[5] Tributary: spot-dancing for elastic services with latency slos. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, 2018), USENIX Association.

[6] Google Kubernetes Engine. https://cloud.google.com/kubernetes-engine/, Accessed October 2019.

[7] Slurm Elastic Computing (Cloud Bursting). `https://slurm.schedmd.com/elastic_computing.html`, Accessed October 2019.

[8] Slurm Workload Manager. https://slurm.schedmd.com/, Accessed October 2019.

[9] Amazon Aws Ec2 Instance Type. https://aws.amazon.com/ec2/instance-types/, Accessed January 2020.

[10] Cloud Computing Market by Service Model. https://www.globenewswire.com/news-release/2020/08/21/2081841/0/en/Cloud-Computing-Industry-to-Grow-from-371-4-Billion-in-2020-to-832-1-Billion-by-2025-at-a-CAGR-of-17-5.html, August 2020.

[11] University of Massachusetts Green High Performance Computing Cluster. `http://wiki.umassrc.org/wiki/index.php/Main_Page`, Accessed August 2020.

[12] Load Sharing Facility. `https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=lsf-foundations`, Accessed May 2021.

[13] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 265–283.

[14] Ai, L., Wu, X., Huang, L., Huang, L., Tang, P., and Li, J. The Multi-shop Ski Rental Problem. In *SIGMETRICS* (June 2014).

[15] Albrecht, J., Tuttle, C., Snoeren, A., and Vahdat, A. Loose Synchronization for Large-scale Networked Systems. In *USENIX ATC* (June 2006).

[16] Alipourfard, O., Liu, H., Chen, J., Venkataraman, S., Yu, M., and Zhang, M. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *NSDI* (March 2017).

[17] Ambati, Pradeep, Bashir, Noman, Irwin, David, Hajiesmaili, Mohammad, and Shenoy, Prashant. Hedge your bets: Optimizing long-term cloud costs by mixing vm purchasing options. In *2020 IEEE International Conference on Cloud Engineering (IC2E)* (2020).

[18] Ambati, Pradeep, Bashir, Noman, Irwin, David, and Shenoy, Prashant. Waiting game: Optimally provisioning fixed resources for cloud-enabled schedulers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2020).

[19] Ambati, Pradeep, Bashir, Noman, Irwin, David, and Shenoy, Prashant. Modeling and analyzing waiting policies for cloud-enabled schedulers. *IEEE Transactions on Parallel and Distributed Systems 32*, 12 (2021), 3081–3100.

[20] Ambati, Pradeep, Goiri, Inigo, Frujeri, Felipe, Gun, Alper, Wang, Ke, Dolan, Brian, Corell, Brian, Pasupuleti, Sekhar, Moscibroda, Thomas, Elnikety, Sameh, Fontoura, Marcus, and Bianchini, Ricardo. Providing slos for resource-harvesting vms in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020).

[21] Ambati, Pradeep, and Irwin, D. Optimizing the cost of executing mixed interactive and batch workloads on transient vms. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2019).

[22] Ambati, Pradeep, Irwin, David E., Shenoy, P., Gao, L., Alieldin, A., and Albrecht, J. Understanding synchronization costs for distributed ml on transient cloud resources. In *2019 IEEE International Conference on Cloud Engineering (IC2E)* (2019).

[23] Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., and Harris, E. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *OSDI* (December 2010).

[24] Babcock, C. Amazon's 'Virtual CPU'? You Figure it Out, in *InformationWeek*, December 23rd 2015.

[25] Barr, J. Amazon Ec2 Beta. AWS Official Blog, August 2006.

[26] Bashir, Noman, Deng, Nan, Rządca, Krzysiek Michał, Irwin, David, Kodakara, Sree, and Jnagal, Rohit. Take it to the limit: Peak prediction-driven resource overcommitment in datacenters. In *EuroSys* (2021).

[27] Beckman, P., Iskra, K., Yoshii, K., and Coghlan, S. The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale. In *Cluster Computing* (September 2006).

[28] Brevik, J., Nurmi, D., and Wolski, R. Predicting Bounds on Queuing Delay for Batch-Scheduled Parallel Machines. In *PPoPP* (March 2006).

[29] Buitinck, Lars, Louppe, Gilles, Blondel, Mathieu, Pedregosa, Fabian, Mueller, Andreas, Grisel, Olivier, Niculae, Vlad, Prettenhofer, Peter, Gramfort, Alexandre, Grobler, Jaques, Layton, Robert, VanderPlas, Jake, Joly, Arnaud, Holt, Brian, and Varoquaux, Gaël. API Design for Machine Learning Software: Experiences from the cikit-learn Project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning* (2013), pp. 108–122.

[30] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, Omega, and Kubernetes. *ACM Queue - Containers 14*, 1 (January-February 2016).

[31] Cui, H., Cipar, J., Ho, Q., Kim, J., Lee, S., Kumar, A., Wei, J., Dai, W., Ganger, G., Gibbons, P., Gibson, G., and Xing, E. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *USENIX ATC* (June 2014).

[32] Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. Remus: High Availability via Asynchronous Virtual Machine Replication. In *NSDI* (April 2008).

[33] Dean, Jeff, and Ghemawatt, Sanjay. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI* (December 2004).

[34] Delimitrou, C., and Kozyrakis, C. HCloud: Resource-Efficient Provisioning in Shared Cloud Systems. In *ASPLOS* (April 2016).

[35] den Bossche, R. Van, Vanmechelen, K., and Broeckhove, J. IaaS Reserved Contract Procurement Optimisation with Load Prediction. *Future Generation Computer Systems 53* (December 2015).

[36] Di, S., Kondo, D., and Wang, C. Optimization and Stabilization of Composite Service Processing in a Cloud System. In *2013 IEEE/ACM 21st International Symposium on Quality of Service (IWQoS)* (June 2013).

[37] Di, S., Wang, C., and Cappello, F. Adaptive Algorithm for Minimizing Cloud Task Length with Prediction Errors. *IEEE Transactions on Cloud Computing 2*, 2 (2014), 194–207.

[38] Erlang, A. K. *On the Rational Determination of the Number of Circuits (1924).* In Life and Works of A K. Erlang , E. Brockmeyer, H J. Halstrom and A. Jensen, Danish Academy of Technical Science, 1948.

[39] Ghit, B., and Epema, D. Better safe than sorry: Grappling with failures of in-memory data analytics frameworks. In *HPDC* (June 2017).

[40] Grandl, Robert, Chowdhury, Mosharaf, Akella, Aditya, and Ananthanarayanan, Ganesh. Altruistic Scheduling in Multi-Resource Clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (2016), OSDI'16.

[41] Guo, T., Sharma, U., Sahu, S., Wood, T., and Shenoy, P. Seagull: Intelligent Cloud Bursting for Enterprise Applications. In *USENIX ATC* (June 2012).

[42] Harlap, A., Tumanov, A., Chung, A., Ganger, G., and Gibbons, P. Proteus: Agile ML Elasticity through Tiered Reliability in Dynamic Resource Markets. In *European Conference on Computer Systems (EuroSys)* (April 2017).

[43] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A., Katz, R., Shenker, S., and Stoica, I. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *NSDI* (March 2011).

[44] Ho, Qirong, Cipar, James, Cui, Henggang, Kim, Jin Kyu, Lee, Seunghak, Gibbons, Phillip B., Gibson, Garth A., Ganger, Gregory R., and Xing, Eric P. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *NIPS* (2013), NIPS.

[45] Hoff, T. High Scalability, The Eternal Cost Savings of Netflix's Internal Spot Market. http://highscalability.com/blog/2017/12/4/the-eternal-cost-savings-of-netflixs-internal-spot-market.html, December 4th 2017.

[46] Hong, Y., Xue, J., and Thottethodi, M. Dynamic Server Provisioning to Minimize Cost in an IaaS Cloud. In *SIGMETRICS* (June 2011).

[47] Hsu, C.J., Nair, Vivek, Freeh, V.W., and Menzies, T. Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM. In *ICDCS* (July 2018).

[48] Hu, M., Luo, J., and Veeravalli, B. Optimal Provisioning for Scheduling Divisible Loads with Reserved Cloud Resources. In *ICON* (December 2012).

[49] Huang, B., Jarrett, N., Babu, S., Mukherjee, S., and Yang, J. Cumulon: Matrix-Based Data Analytics in the Cloud with Spot Instances. *Proceedings of the VLDB Endowment (PVLDB) 9*, 3 (November 2015).

[50] Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., and Goldberg, A. Quincy: Fair Scheduling for Distributed Computing Clusters. In *SOSP* (October 2009).

[51] Jensen, A. *Moe's Principle: An Econometric Investigation Intended as an Aid in Dimensioning and Managing Telephone Plant.* The Copenhagen Telephone Company, 1950.

[52] Kadupitige, J., Jadhao, V., and Sharma, P. Modeling the Temporally Constrained Preemptions of Transient Cloud VMs. In *HPDC* (June 2020).

[53] Kuchnik, Michael, Park, J., Cranor, C., Moore, Elisabeth, DeBardeleben, Nathan, and Amvrosiadis, George. This is why ml-driven cluster scheduling remains widely impractical.

[54] Li, Mu, Andersen, David, Park, Jun Woo, Smola, Alexander, Ahmed, Amr, Josifovski, Vanja, Long, James, Shekita, Eugene, and Su, Bor-Yiing. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI* (November 2014).

[55] Liu, L., and Kulkarni, V. Balking and Reneging in M/G/s Systems: Exact Analysis and Approximations. *Probability in the Engineering and Informational Sciences 22*, 3 (July 2008).

[56] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[57] Niu, S., Zhai, J., Ma, X., Tang, X., and Chen, W. Cost-effective Cloud HPC Resource Provisioning by Building Semi-Elastic Virtual Clusters. In *SC* (November 2013).

[58] Nurmi, D., Brevik, J., and Wolski, R. QBETS: Queue Bounds Estimation from Time Series. In *JSSPP* (June 2007).

[59] Omer, S., N.Yigitbasi, Iosup, A., and Epema, D. Trace-based Evaluation of Job Runtime and Queue Wait Time Predictions in Grids. In *HPDC* (June 2009).

[60] Park, Jun Woo, Tumanov, Alexey, Jiang, Angela, Kozuch, Michael A., and Ganger, Gregory R. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth EuroSys Conference* (2018).

[61] Paszke, Adam, Gross, Sam, Massa, Francisco, Lerer, Adam, Bradbury, James, Chanan, Gregory, Killeen, Trevor, Lin, Zeming, Gimelshein, Natalia, Antiga, Luca, Desmaison, Alban, Kopf, Andreas, Yang, Edward, DeVito, Zachary, Raison, Martin, Tejani, Alykhan, Chilamkurthy, Sasank, Steiner, Benoit, Fang, Lu, Bai, Junjie, and Chintala, Soumith. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems* (2019).

[62] Petrini, F., Kerbyson, D.J., and Pakin, S. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on 8,192 Processors of ASCI Q. In *SC* (November 2003).

[63] Sharma, P., Guo, T., He, X., Irwin, D., and Shenoy, P. Flint: Batch-Interactive Data-Intensive Processing on Transient Servers. In *European Conference on Computer Systems (EuroSys)* (April 2016).

[64] Sharma, P., Irwin, D., and Shenoy, P. Portfolio-driven Resource Management for Transient Cloud Servers. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (June 2017).

[65] Sharma, P., Lee, S., Guo, T., Irwin, D., and Shenoy, P. SpotCheck: Designing a Derivative IaaS Cloud on the Spot Market. In *European Conference on Computer Systems (EuroSys)* (April 2015).

[66] Shastri, S., Rizk, A., and Irwin, D. Transient Guarantees: Maximizing the Value of Idle Cloud Capacity. In *SC* (November 2016).

[67] Shen, S., Deng, K., Iosup, A., and Epema, D. Scheduling Jobs in the Cloud using On-demand and Reserved Instances. In *Euro-Par* (August 2013).

[68] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The Hadoop Distributed File System. In *MSST* (May 2010).

[69] Singh, R., Irwin, D., Shenoy, P., and Ramakrishnan, K.K. Yank: Enabling Green Data Centers to Pull the Plug. In *Symposium on Networked Systems Design and Implementation (NSDI)* (April 2013).

[70] Singh, R., Sharma, P., Irwin, D., Shenoy, P., and Ramakrishnan, K.K. Here Today, Gone Tomorrow: Exploiting Transient Servers in Datacenters. *IEEE Internet Computing 18*, 4 (April 2014).

[71] Singh, Rahul, Sharma, Prateek, Irwin, David, Shenoy, Prashant, and Ramakrishnan, K.K. Here Today, Gone Tomorrow: Exploiting Transient Servers in Data Centers. *IEEE Internet Computing 18*, 4 (July 2014).

[72] Smith, W., Taylor, V., and Foster, I. Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance. In *JSSPP* (April 1999).

[73] Souza, Abel, Pelckmans, Kristiaan, Ghoshal, Devarshi, Ramakrishnan, Lavanya, and Tordsson, Johan. ASA - The Adaptive Scheduling Architecture. In *HPDC* (June 2020).

[74] Subramanya, S., Guo, T., Sharma, P., Irwin, D., and Shenoy, P. SpotOn: A Batch Computing Service for the Spot Market. In *Symposium on Cloud Computing (SoCC)* (August 2015).

[75] Takacs, L. *Introduction to the Theory of Queues*. Oxford University Press, 1962.

[76] Tirmazi, M., Barker, A., Deng, N., Haque, M., Qin, Z., Hand, S., Harchol-Balter, M., and Wilkes, J. Borg: The Next Generation. In *EuroSys* (April 2020).

[77] Tumanov, A., Jiang, A., Park, J., Kozuch, M., and Ganger, G. Jamaisvu: Robust Scheduling with Auto-Estimated Job Runtimes, Accessed September 2016.

[78] Tumanov, A., Zhu, T., Park, J., Kozuch, M., Harchol-Balter, M., and Ganger, G. TetriSched: Global Rescheduling with Adaptive Plan-Ahead in Dynamic Heterogeneous Clusters. In *EuroSys* (March 2016).

[79] Valiant, L. A Bridging Model for Parallel Computation. *CACM 33*, 8 (August 1990).

[80] Venkataraman, S., Yang, Z., Franklin, M., Recht, B., and Stoica, I. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI* (March 2016).

[81] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale Cluster Management at Google with Borg. In *European Conference on Computer Systems (EuroSys)* (April 2015).

[82] Wang, C., Urgaonkar, B., Nasiriani, N., and Kesidis, G. Using Burstable Instances in the Public Cloud: Why, When, and How? *ACM on Measurement and Analysis of Computing Systems 1*, 1 (June 2017).

[83] Wang, W., Li, B., and Liang, B. To Reserve or Not to Reserve: Optimal Online Multi-Instance Aquisition in IaaS Clouds. In *ICAC* (June 2013).

[84] Wang, W., Niu, D., Li, B., and Liang, B. Dynamic Cloud Resource Reservation via Cloud Brokerage. In *ICDCS* (July 2013).

[85] Wang, Z., Gao, L., Gu, Y., Bao, Y., and Yu, G. FSP: Towards Flexible Synchronous Parallel Framework for Expectation-Maximization based Algorithms on Cloud. In *SoCC* (September 2017).

[86] Whitt, Ward. Erlang B and C Formulas: Problems and Solutions. http://www.columbia.edu/ ww2040/ErlangBandCFormulas.pdf, 2002.

[87] Xu, Z., Stewart, C., Deng, N., and Wang, X. Blending On-Demand and Spot Instances to Lower Costs for In-Memory Storage. In *International Conference on Computer Communications (Infocom)* (July 2016).

[88] Yadwadkar, N., Hariharan, B., Gonzalez, J., Smith, B., and Katz, R. Selecting the Best VM across Multiple Public Clouds: A Data-driven Performance Modeling Approach. In *SoCC* (September 2017).

[89] Yan, Y., Gao, Y., Guo, Z., Chen, B., and Moscibroda, T. TR-Spark: Transient Computing for Big Data Analytics. In *Symposium on Cloud Computing (SoCC)* (October 2016).

[90] Yang, Y., Kim, G., Song, W., Lee, Y., Chung, A., Qian, Z., Cho, B., and Chun, B. Pado: A Data Processing Engine for Harnessing Transient Resources in Datacenters. In *European Conference on Computer Systems (EuroSys)* (April 2017).

[91] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI* (April 2012).

[92] Zaharia, M., Konwinski, A., Joseph, A., Katz, R., and Stoica, I. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI* (December 2008).

[93] Zhao, G., Gao, L., and Irwin, D. Sync-on-the-fly: A Parallel Framework for Gradient Descent Algorithms on Transient Resources. In *BigData* (December 2018).

[94] Zheng, L., Joe-Wong, C., Brinton, C., Tan, C., Ha, S., and Chiang, M. On the Viability of a Cloud Virtual Service Provider. In *SIGMETRICS* (June 2016).

[95] Zheng, L., Joe-Wong, C., Tan, C., Chiang, M., and Wang, X. How to Bid the Cloud. In *ACM SIGCOMM Conference (SIGCOMM)* (August 2015).